(continued from part 16)

Integrated circuit counters

As we know from *Digital Electronics 13*, flip-flops can be used to build counters which divide by (i.e. count in cycles of) 2, 4, 8, 16 and higher multiples of two, simply by connecting the required number of flip-flops in series. As the **modulus** of a counter is the number of states the counter counts through before returning to its initial state, such counters are known as modulo-2, modulo-4, modulo-8, modulo-16 counters, and so on.

Using flip-flops, we can also construct a counter with a modulus which is not a multiple of 2 by steering the counter back to its initial state before the natural modulus cycle is completed. Some ready-built counters also exist in the 7400 series which can be turned into counters having a different modulus simply by altering connections or by the addition of a few simple components.

In this chapter we will learn how to make counters with any modulus between 2 and 14; counters for numbers greater than this can generally be constructed by connecting other counters in series. For example, a modulo-18 counter is made by connecting a modulo-2 counter and a modulo-9 counter in series; modulo-48 counters are made from modulo-6 and modulo-8 counters in series, and so on.

The counter ICs we shall examine (the 7490 modulo-10 counter, the 7492 modulo-12 counter, and the 7493 modulo-16 counter) are all constructed internally from four J-K master-slave flip-flops and so have corresponding outputs Q_A, Q_B, Q_C and Q_D. The count sequences of each IC counter are shown as function tables in *tables 2-4*, where the logic states at each output are shown against each stage of the count. All three ICs are configured in two parts to obtain the maximum modulus: the 7490 IC counter is formed from a modulo-2 counter followed by a modulo-5 counter; the 7492 IC is formed from a modulo-2 counter and a modulo-6 counter; the 7493 IC is a modulo-2 followed by a modulo-8 counter.

All of the counters have reset inputs, which can be used to reset the counter

Table 2
Count sequence of a 7490 IC counter

Count stage	Outputs			
	Q_D	Q_C	Q_B	Q_A
0	L	L	L	L
1	L	L	L	H
2	L	L	H	L
3	L	L	H	H
4	L	H	L	L
5	L	H	L	H
6	L	H	H	L
7	L	H	H	H
8	H	L	L	L
9	H	L	L	H

Table 3
Count sequence of a 7492 IC counter

Count stage	Outputs			
	Q_D	Q_C	Q_B	Q_A
0	L	L	L	L
1	L	L	L	H
2	L	L	H	L
3	L	L	H	H
4	L	H	L	L
5	L	H	L	H
6	H	L	L	L
7	H	L	L	H
8	H	L	H	L
9	H	L	H	H
10	H	H	L	L
11	H	H	L	H

Table 4
Count sequence of a 7493 IC counter

Count stage	Outputs			
	Q_D	Q_C	Q_B	Q_A
0	L	L	L	L
1	L	L	L	H
2	L	L	H	L
3	L	L	H	H
4	L	H	L	L
5	L	H	L	H
6	L	H	H	L
7	L	H	H	H
8	H	L	L	L
9	H	L	L	H
10	H	L	H	L
11	H	L	H	H
12	H	H	L	L
13	H	H	L	H
14	H	H	H	L
15	H	H	H	H

back to its initial state.

Modulo-2 counter

As all of the counter ICs have an internal modulo-2 counter, it is a simple task to make such a counter from any of the ICs. *Figure 7* shows an example circuit of a 7490 IC used in this way. Pin 12, the Q_A output (i.e. the output of the first flip-flop), forms the circuit output. Pin 2, the circuit's reset control must be at logic 0 when the circuit is counting. Opening the switch allows pin 2 to go to logic 1 and so the counter is reset – the output from pin 12 goes to logic 0 whatever input is applied to pin 14. Any circuit giving TTL compatible output signals may be used in place of the resistor (known as a **pull-up resistor**) and switch.

Modulo-3 counter

Because the IC cannot be configured as a counter with a natural modulus of three, the counter must be steered back to the initial stage, on the fourth stage of the count.

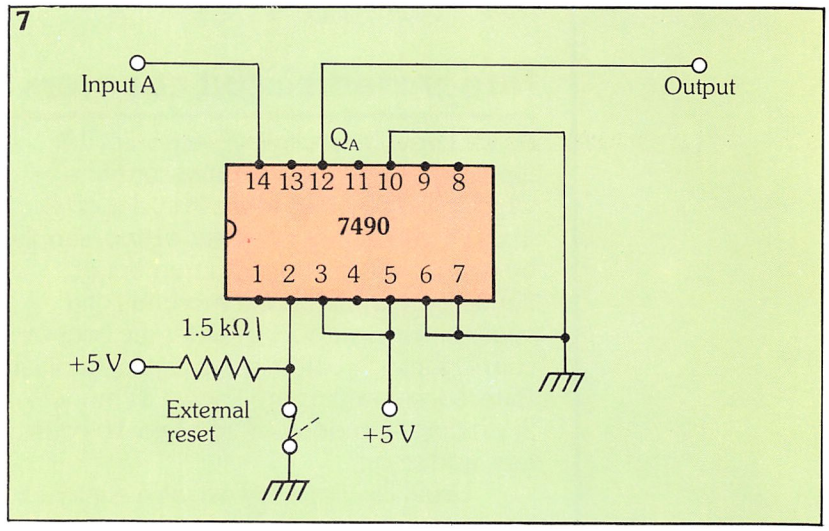
From *table 2*, we can see that the fourth stage of the count, i.e. 3, causes the Q_A and Q_B outputs to be high. Outputs Q_A and Q_B are at pins 12 and 9, and the NAND gate circuits of gates 1 and 2 detect the high states and reset the counter at the fourth stage. NAND gate 3 is used to provide an external reset facility as shown in *figure 8*.

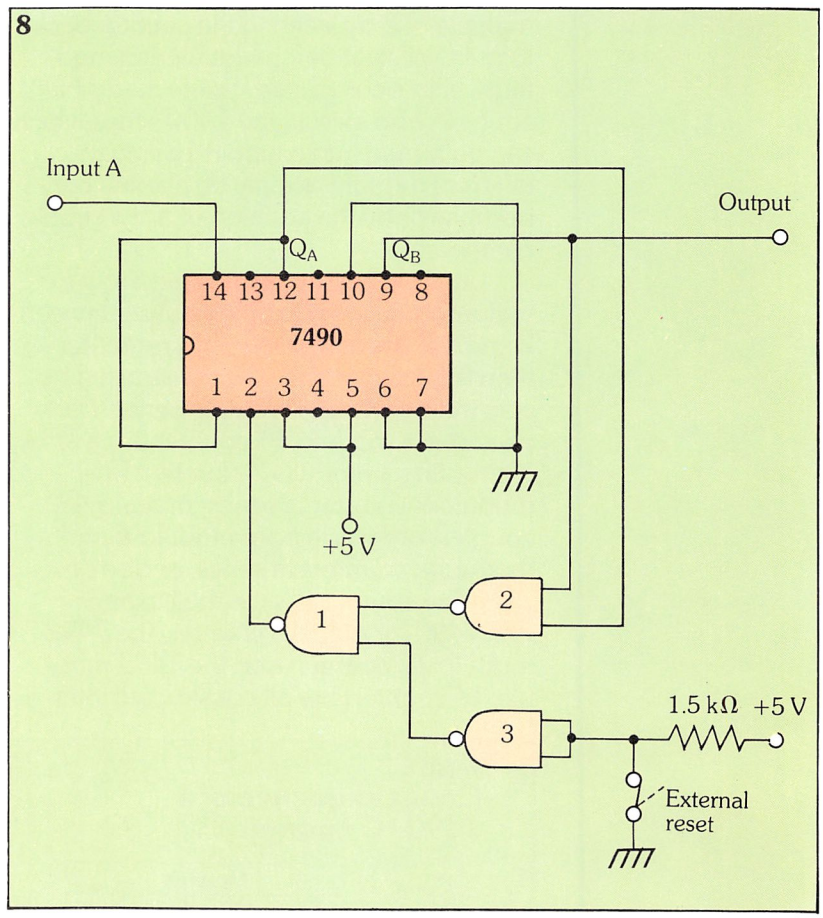
If an external reset facility is not required, the circuit shown in *figure 9* may be used. Here, the Q_A output of the counter is connected to reset pin 2, and the Q_B output is connected to a second reset, pin 3. When both Q_A and Q_B are high (at the fourth count stage) the counter is reset.

Modulo-4 counter

A 7493 IC may be used for this purpose and is connected as shown in *figure 10*. The second and third flip-flops are used to give a counter with a natural modulus of 4, so the input is applied to pin 1 – input B, of the second flip-flop. Consequently the output is from Q_C, at pin 8.

Reset pin for this IC is also pin 2 and the earlier remarks regarding external reset of the modulo-2 counter also apply to this circuit.





Modulo-5 counter

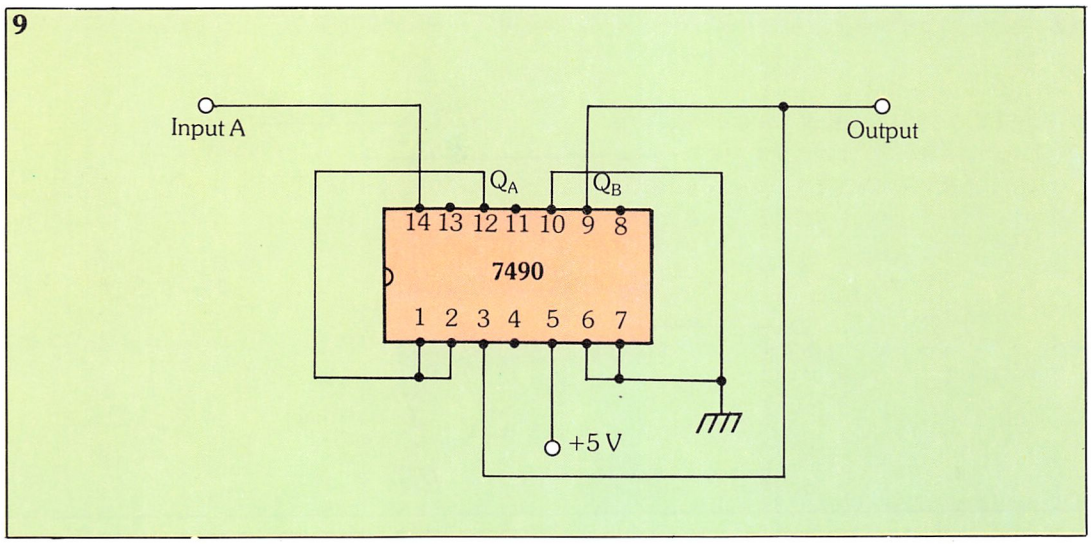
The second part counter of the 7490 IC has a natural modulus of five and the circuit is shown in *figure 11*.

Input B is used (pin 1) and output Q_D (pin 11), so flip-flops B, C, and D form the counter. The usual external reset facility to reset the counter back to its initial state is included.

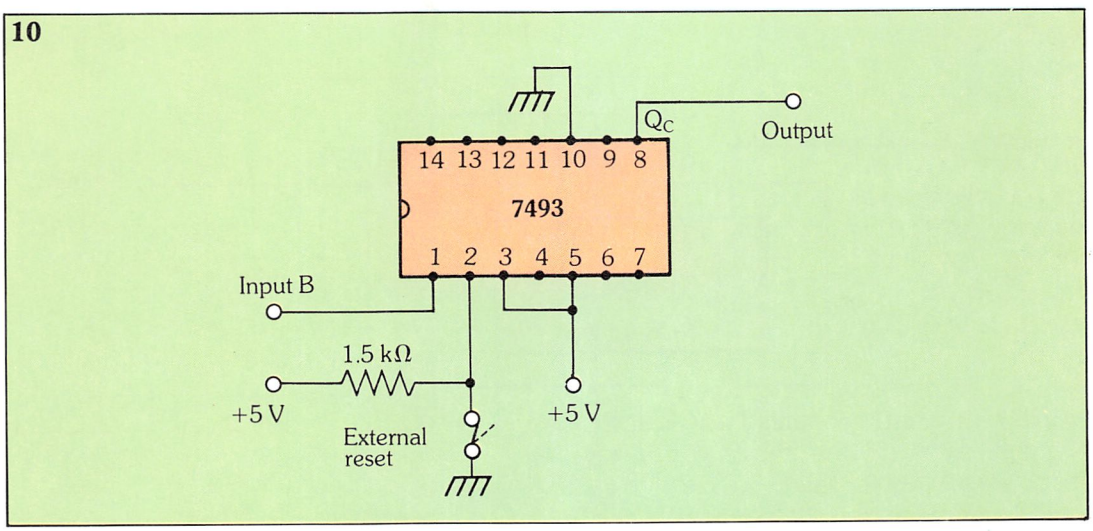
7. 7490 IC used as a modulo-2 counter.

8. 7490 IC used as a modulo-3 counter with external reset.

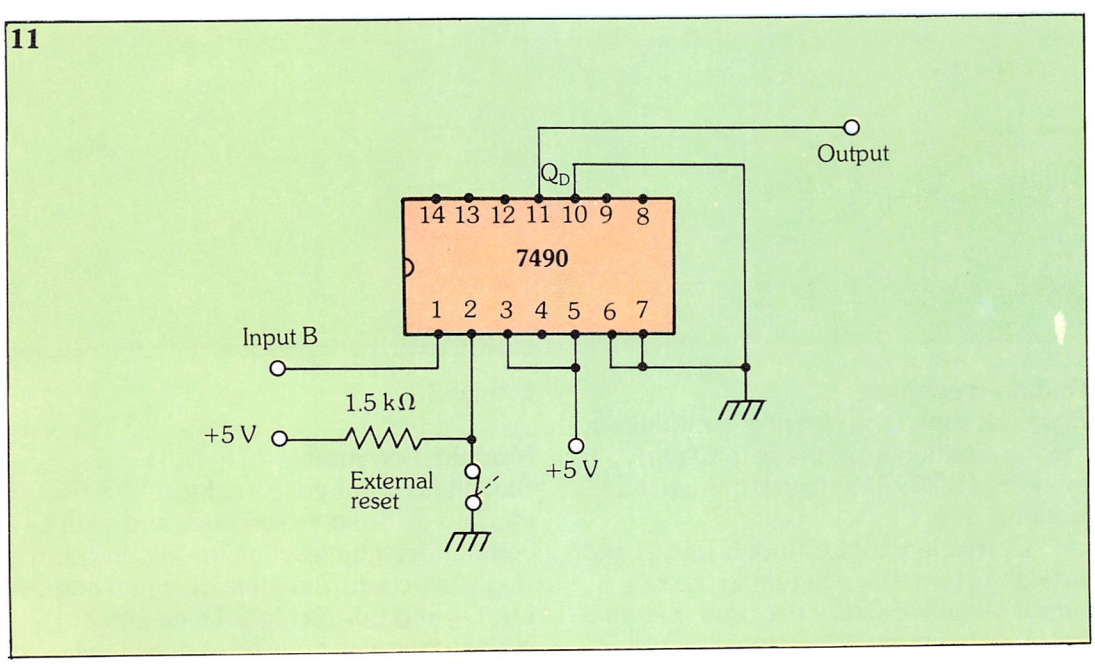
9. 7490 IC without the external reset.

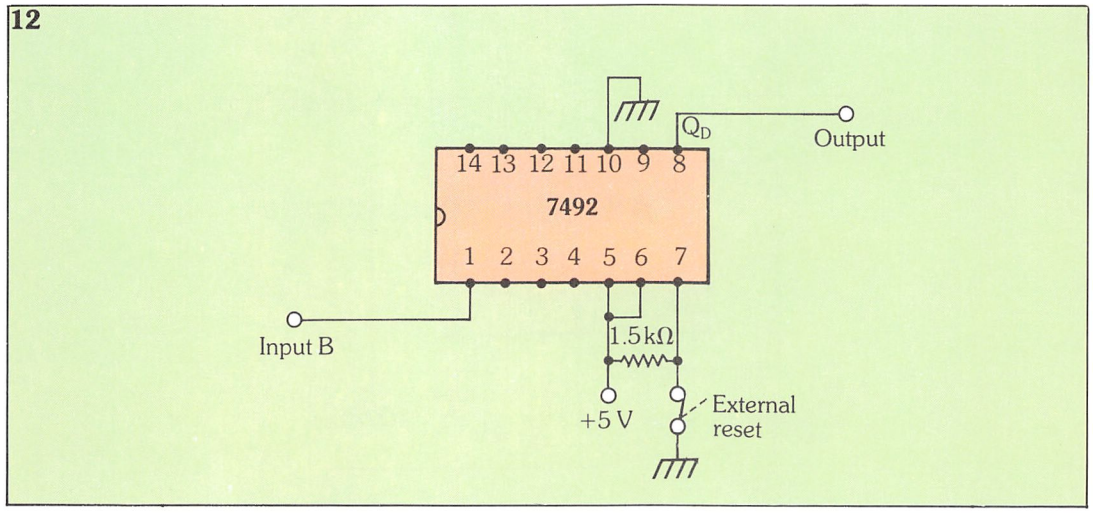


10. 7493 IC used as a modulo-4 counter.

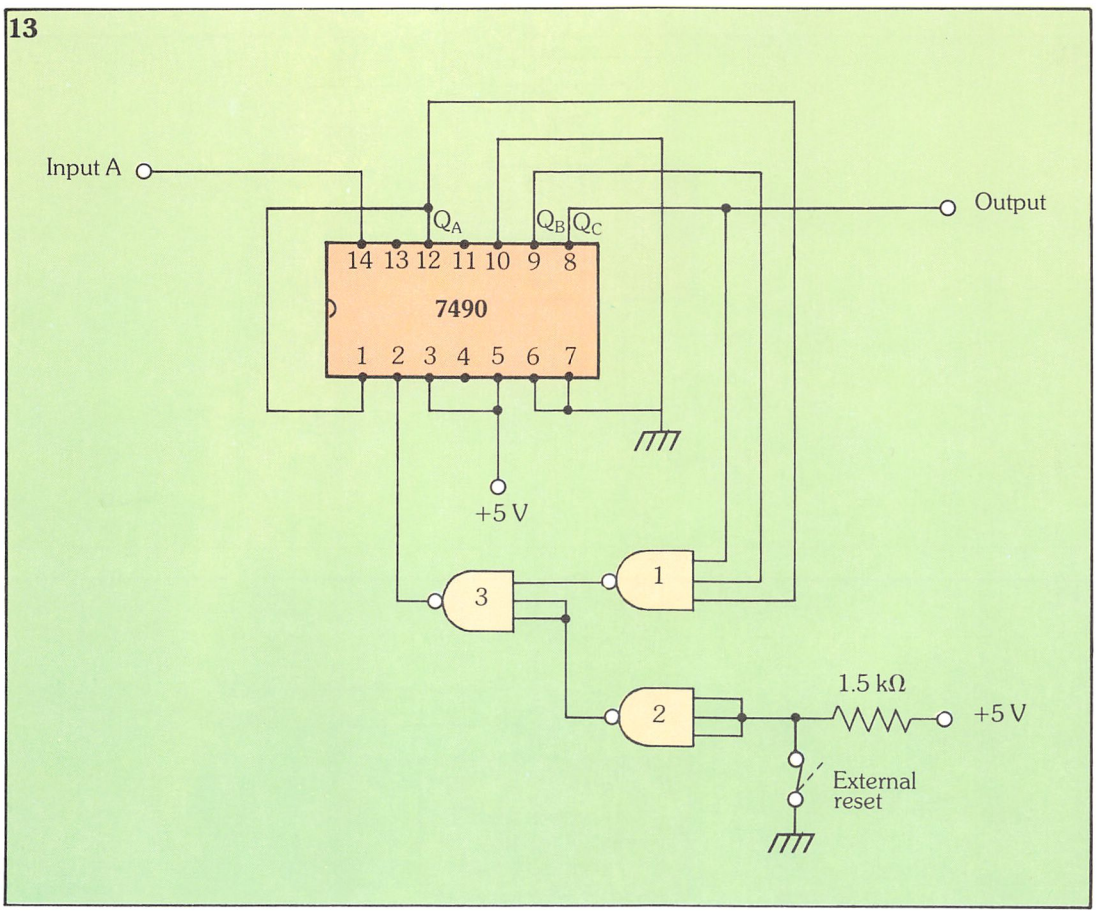


11. 7490 IC used as a modulo-5 counter.





12. A modulo-6 counter formed from the second part counter of a 7492 IC.



13. 7490 IC modulo-7 counter.

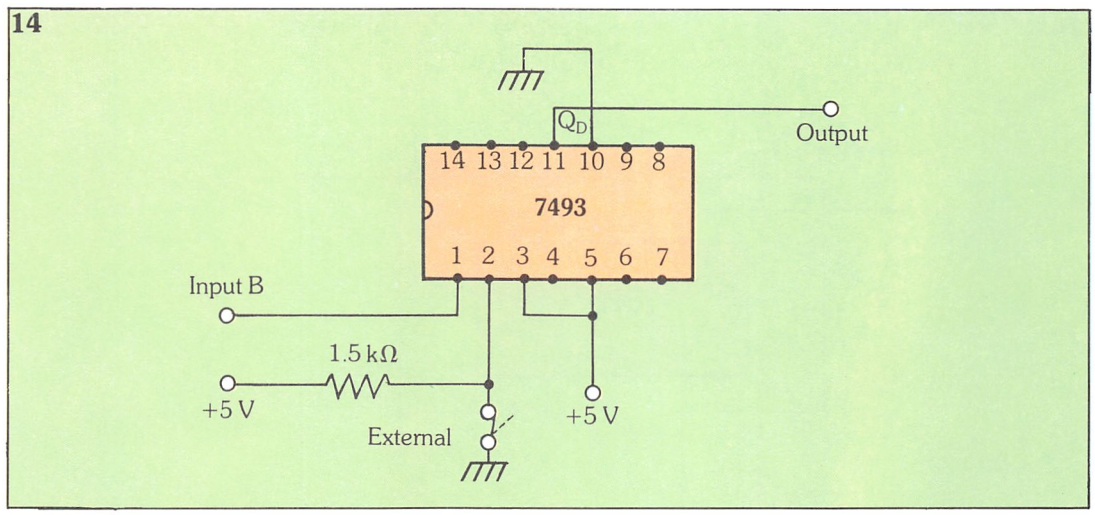
Modulo-6 counter

Figure 12 shows the circuit of a modulo-6 counter, formed from the second part counter of a 7492 IC, which has a natural modulus of 6.

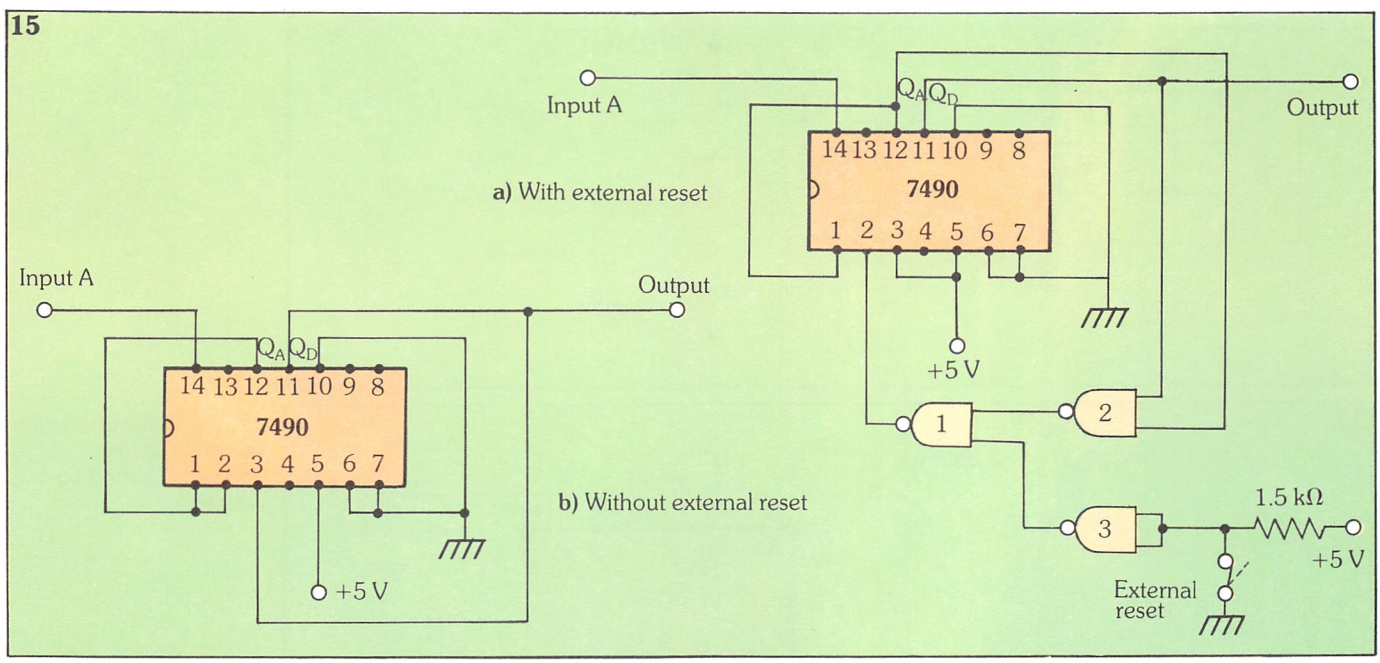
Correspondingly, input B (pin 1) and output Q_D (pin 8) – different to the Q_D output of the 7490 IC – are used. External reset facility operates by connecting pin 7 to logic 1.

Modulo-7 counter

Steering is required to enable a 7490 IC modulo-7 counter to be built, and so the counter must be reset on the eighth count stage (i.e. count 7 in *table 2*), when outputs Q_A, Q_B and Q_C are high. Three input NAND gates can be used to detect the



14. Input B and output Q_D of a 7493 IC form a modulo-8 counter.



15. 7490 IC forming a modulo-9 counter: (a) with external reset; **(b)** without reset.

three high logic states and reset the counter to its initial count stage, as shown in *figure 13*.

Modulo-8 counter

Second part of a 7493 IC has a natural modulus of 8, so input B (pin 1) of the IC and output Q_D (pin 11) may be used to form a modulo-8 counter. The circuit is shown in *figure 14*.

Modulo-9 counter

Like the modulo-3 counter, two circuits can be used for a modulo-9 counter, depending on whether an external reset facility is required. The tenth count stage, i.e. 9, causes outputs Q_A and Q_B to be high and it is a simple job to detect this using NAND gates as in *figure 15a* and reset the counter.

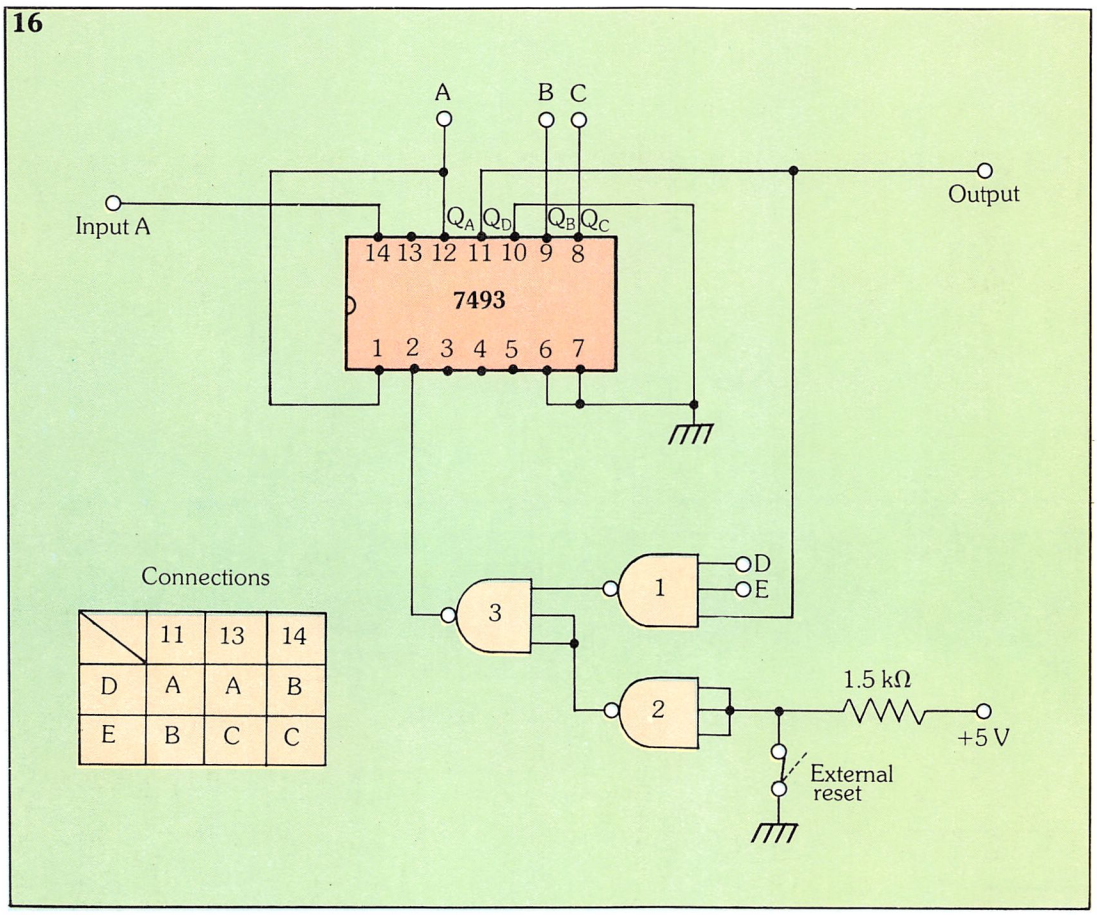
Figure 15b (without external reset facility) shows how the two high logic states of outputs Q_A and Q_D reset the counter via reset pins 2 and 3.

Modulo-10 counter

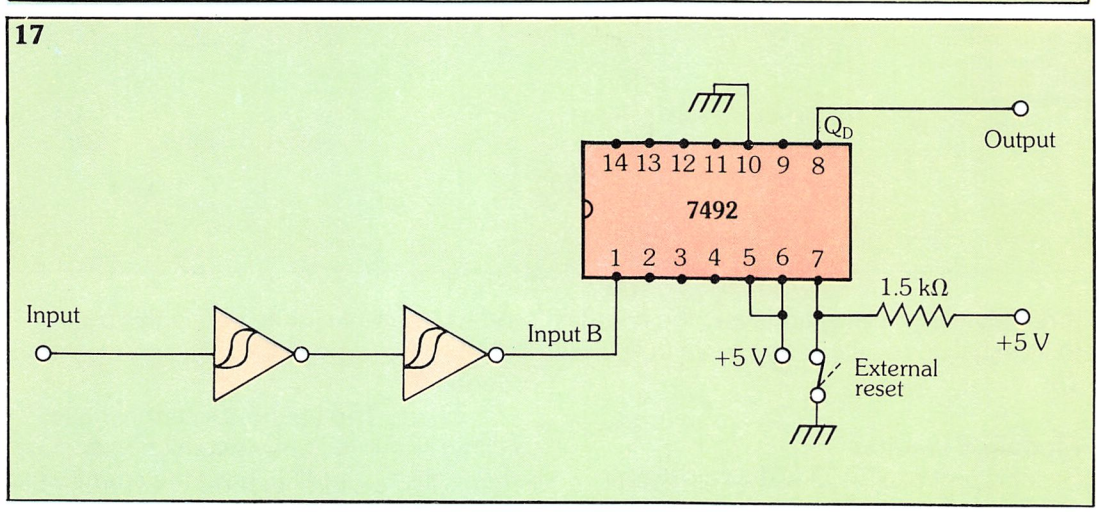
A 7490 IC can be used here, because it has a natural modulus of 10, using input A and output Q_D. Similarly, a 7492 IC can be used as a natural modulo-12 counter.

Modulo-11, 13 and 14 counters

These counters are grouped together because they all use the 7493 IC counter –



16. Main circuit for modulo-11, modulo-13 and modulo-14 counters formed from the 7493 IC.



17. Two Schmitt trigger inverters used with a 7492 IC in a modulo-6 counter.

only their connections differ. *Figure 16* illustrates the main circuit and a table corresponding to the extra connection(s) required. For example, if a modulo-13 counter is required, point A should be connected to point D, and point C should be connected to point E.

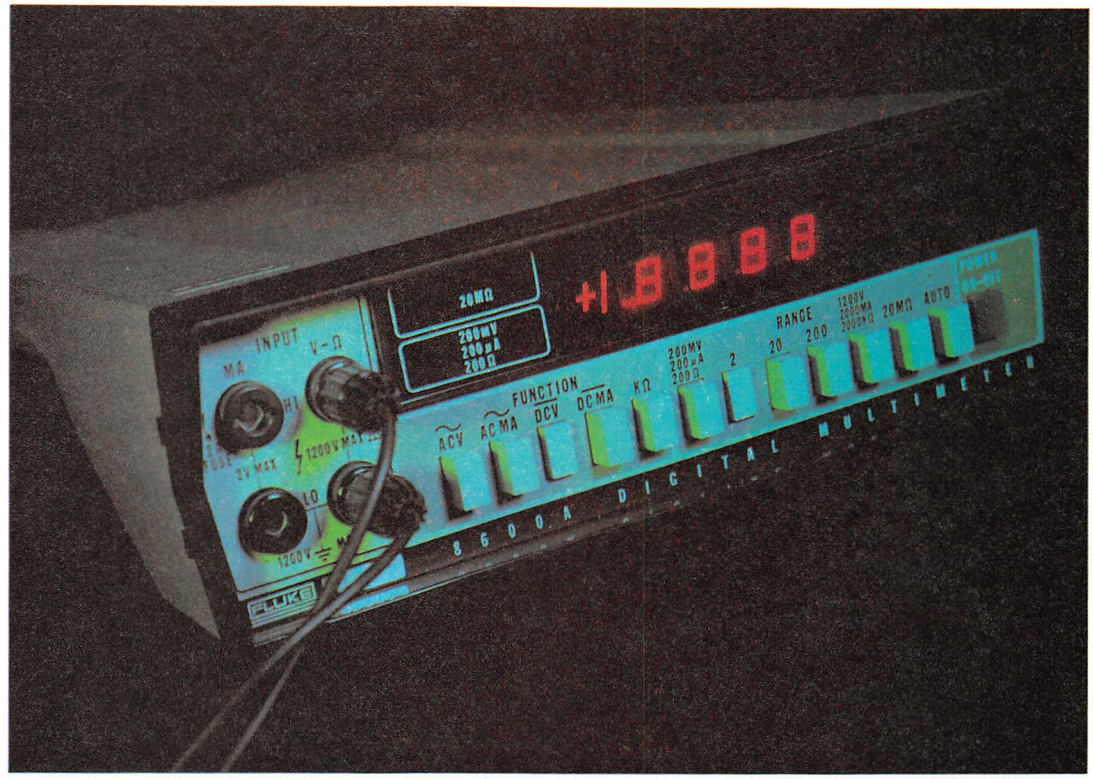
Use of Schmitt triggers

The counters that we have studied here are only suitable for use with input signals having fast, i.e. steep, positive and negative edges. Input signals with slow edges might cause the counter to falter between its input's high and low states, thereby producing errors in the counts.

This situation can be avoided by including a circuit, known as a **Schmitt trigger**, before the counter input. This switches from one logic state to the other

much faster than the slow edges of the input signal, resulting in very fast edges to the signal applied to the counter, almost regardless of the input waveform. *Figure 17* shows how two Schmitt trigger inverters, for example from a 7414 hex Schmitt trigger inverter IC, can be used with a 7492 IC in a modulo-6 counter.

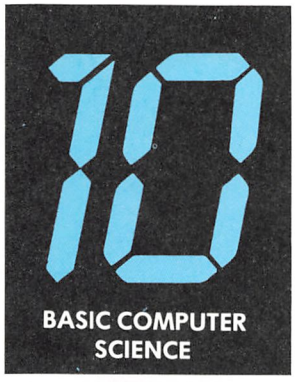
Right: a digital multimeter, using TTL ICs incorporating counters.



Paul Brierley

Glossary

level-triggered	latch operation when an input data change, made during the clock pulse, changes the latch output
modulus	the number of states a counter counts through before returning to its original state. Written as 'modulo' when used as a prefix, e.g. modulo-12 counter
negative-edge triggered	latch operation when an input data change, made at the same time as the clock pulse's negative-edge, changes the latch output
positive-edge triggered	operation of a latch, in which the output changes corresponding to a data change at the input made at the same time as the clock pulse's positive-edge
pull-up resistor	resistor connected between +V and, say, a gate input, which holds the input voltage of the gate at logic 1
Schmitt trigger	type of circuit which can be used to provide a signal with fast positive and negative edges, from a signal with slow positive and negative edges



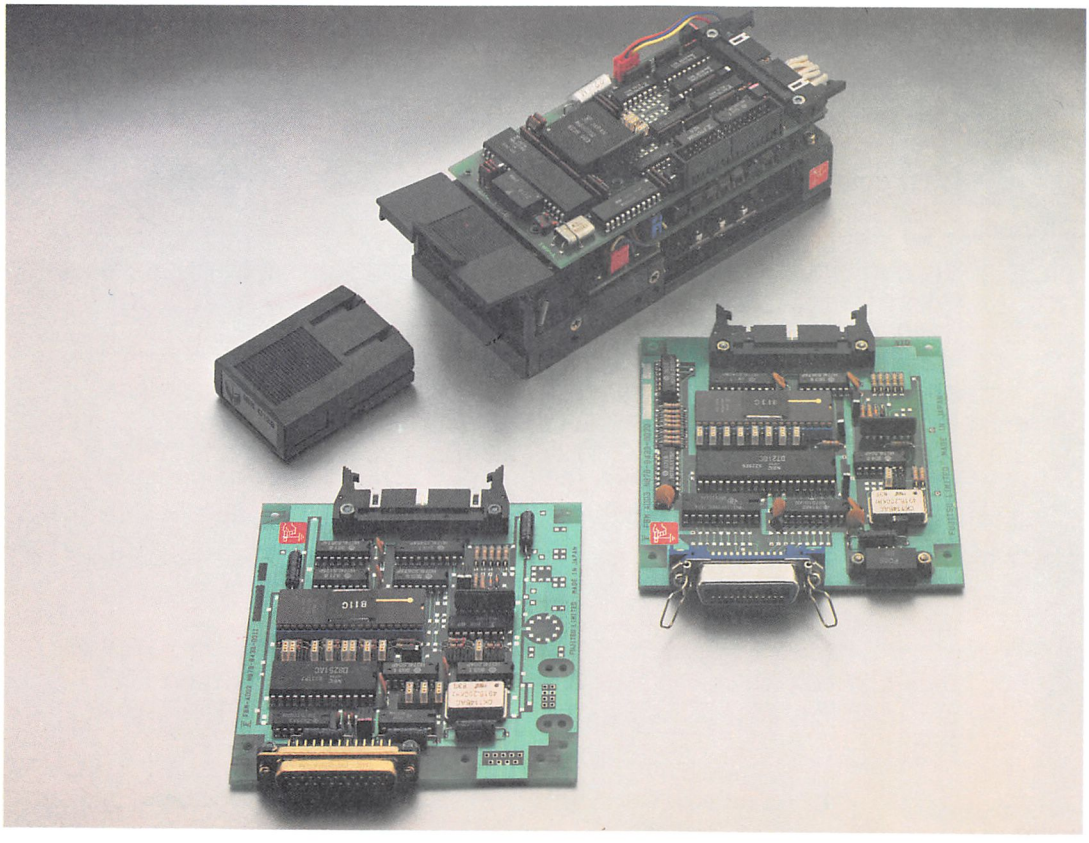
Operating systems -an overview

What is an operating system

The software that enables a computer to function can be broadly divided into two different types: the operating system, comprising the **executive** (that part of the operating system that resides in memory) and its **utilities** (for example, a compiler or a text editor), sometimes referred to as systems software; and applications software.

Applications software comprises programs written to perform specific tasks for the user, e.g. word processing, typesetting and computer-aided design packages, and is usually written in a high level language. (There are exceptions, however, many computer games, for example, are written in assembly language or machine code.) This applications software, however, cannot function alone and needs an operating system to support it. It is usually said to 'run under' the host operating system.

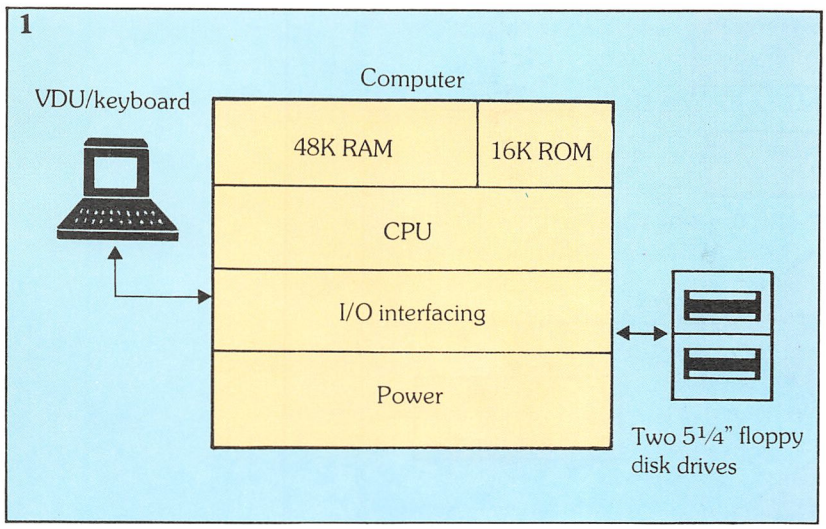
An **operating system** is a set of interrelated programs provided by a computer manufacturer. Although there are many differences between systems, operating systems often provide the following: resource management; protection (preventing unauthorised access to other users files, and data from being corrupted); keeps track of time and resources used for different jobs and maintains schedules (mainframes); and eases the burden of applications program development. As an analogy, take a large office building: the business of the company occupying the



Left: the IBS 1 Mbit bubble cassette memory system (background) with serial and parallel interface cards (foreground). (Photo: Immediate Business Systems).

building can be thought of as the application; the services, such as lifts, security, heating etc. can be thought of as the operating system.

The relationship between the applications software and the operating system continually passing control between each other.



1. A personal computer system.

It is important to remember that the *kinds* of programs that are written as part of the applications software for one computer, could actually form part of the operating system of another – there are no absolute distinctions between the two.

The fundamental function of the operating system is to interface the user programs to the physical hardware of the machine. Thus, the programes can be (to some extent) hardware independent.

We will now go on to investigate the general development and functioning of operating systems. Resource management, one of the most important functions of the operating system, will be discussed in detail in *Basic Computer Science 11*.

An operating system on a personal computer

The size and complexity of a computer's operating system depends upon the functions the computer was designed to perform. We'll take, as an example, a personal computer, and we'll examine how the machine executes a program written in BASIC, followed by a program written in FORTRAN. We will then compare this with the way the same two programs are run on a large mainframe computer.

The personal computer system shown in *figure 1* has two mini floppy disk drives, a keyboard, VDU and 64K of semiconductor memory divided into 48K of RAM and 16K of ROM. The computer has been divided into four separate modules: memory; CPU (containing the CPU and its related circuitry); I/O interfacing (providing the electrical interfacing between the system busses and I/O devices – it also contains an I/O processor for the floppy disks); and the power supply.

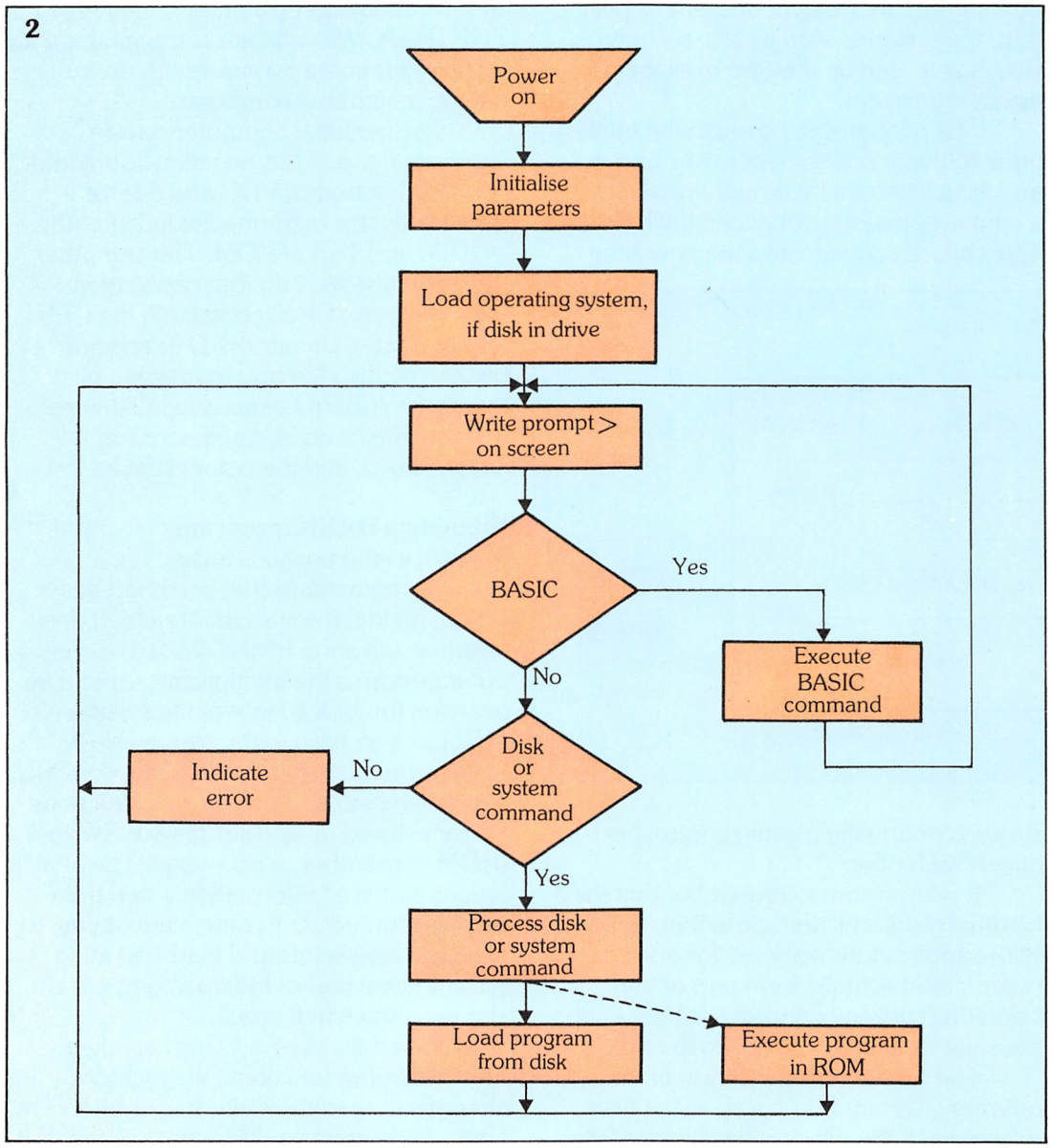
Running a BASIC program

As soon as the power is turned on, a **prompt** character flashes on the VDU indicating that the system is ready. If, for instance, we enter PRINT 2 * 3, the computer performs the multiplication and 6 appears on the VDU. How did this happen?

Let's go back to the beginning. As soon as the power is switched on, the CPU immediately executes a set of instructions that are stored in a preset zone of ROM. (ROM, remember, is non-volatile i.e. it retains its stored information when the power is turned off.) These instructions, to initialise parameters and load operating system, form part of the operating system and are shown in *figure 2*.

After initialisation (which involves **housekeeping** functions) the machine attempts to read the disk (if it has one.) The operating system – which contains the disk drive controller – is kept on disk because its routines are too big to store in ROM: one advantage of this independence is that the operating system can be more easily updated to a new version. (Machines which do not have disks, such as the Sinclair ZX81, have their operating systems stored elsewhere, such as ROM or cassette.)

The operating system routines are always stored at a specific location on the disk (the address for this is stored in ROM) and are loaded into a special area of RAM (reserved for system use only) under the control of instructions in ROM. So, if a disk unit is attached to the computer system, a set sequence of events occurs: first, an instruction is sent to ROM to find the

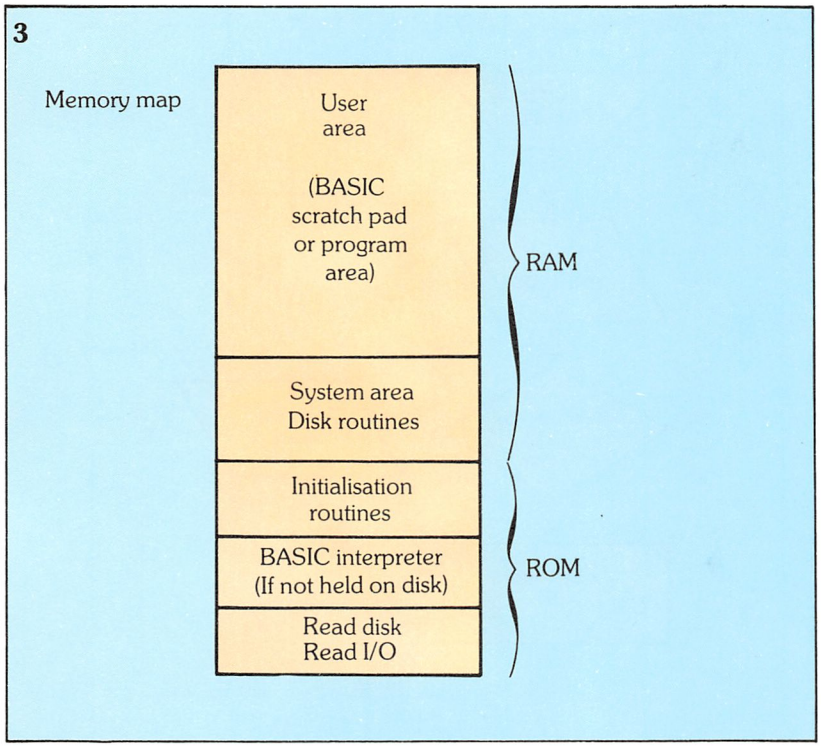


2. Flowchart for operating system instructions when the power is switched on.

address of the operating system routine; second, when the routine is located, it is read into the special area of RAM. Thereafter, other information can be written onto or read from the disk(s). This process is referred to as **bootstrapping** (or 'booting') as the operating system has 'pulled itself up by its bootstraps'.

The operating system has been configured (either by the user or the manufacturer) to search the disk for the BASIC interpreter (the program that translates each BASIC command into machine code). Upon finding this, it will be loaded into memory and control is passed to it. After the interpreter has itself initialised, it displays the system prompt indicating to the user that it is now ready to accept input. (Again, if the computer has no disk unit, then the BASIC interpreter will be stored in ROM.)

Once a set of characters is entered at the keyboard and the RETURN or ENTER key is pressed, the characters are scanned. Should they be a BASIC command without a line number, such as PRINT 2 * 3, the command is immediately executed and control is returned to the keyboard; if it is deferred, i.e. a command with a line number, then it is stored in a scratchpad memory area. *Figure 3* illustrates a memory map for our personal computer.



3. Memory map for the personal computer showing scratchpad area.

If the command from the keyboard specifies the loading and running of a program, then the program is loaded into the system's memory from disk and then has control passed to it for execution.

Running a FORTRAN program

Now suppose that we want to run a FORTRAN program that has been written and stored on disk. In order for the program to be able to run, it has to be compiled (i.e. translated into machine language by the compiler). The operating system loads the FORTRAN compiler (from disk) and then hands over control to it. The FORTRAN compiler then reads the FORTRAN program and generates a **binary file** of corresponding machine language instructions which is stored on disk by the operating system. The machine language instructions are then loaded into RAM, and the program is ready to run.

The important difference to note between running a FORTRAN and a BASIC program is that whereas the BASIC interpreter translates the program into machine code line by line and runs it immediately, the compiler requires the whole program to be input before it is checked and errors identified – and only then is the binary file generated and the program run.

Batch processing

The first generation of computers either had very simple operating systems or none at all. Computers could only be used by one person at a time to write, debug and run their program; all input and output functions were directly controlled by the user's program which was written in either assembly code or machine language. As the number of computer applications increased, this kind of operation became too expensive. In order to alleviate some of these problems, routines were developed to aid the I/O proccesses and **batch processing** was introduced so that the machine could be used more efficiently. This was the beginning of the development of the operating system.

During batch processing, the system reads in one job, passes control to it and then, when it is complete, reads in the next job and so on. Instead of each user operating the machine separately, one person, the **computer operator**, controls the collection and loading of all jobs. Batch processing programs and their corresponding data are generally held on disk or tape. The operating system takes each job in turn, according to the priority assigned to it.

The wide variety of second generation machines resulted in the need for more and more system functions. Input/output requirements became varied, as did the different I/O devices. Early batch systems used to call the operating system routines from auxiliary memory when they were needed. However, as programs came to depend increasingly on system routines, it became more practical to retain these in central memory. Such operating systems are known as **resident monitors**.

The functions of the batch control program and the resident monitor are fairly similar, although the latter is more sophisticated. Both contain a collection of routines which oversee the loading and running of user programs. This task consists of two basic functions: **executive control** and **I/O control**.

Executive control

The executive control routines ensure the availability of the resources required by each job. The user has to specify the

language translator needed, the amount of memory space needed, time requirements, the I/O devices required, and job accounting information.

The user informs the operating system of its required operating parameters by utilising a special set of commands called **job control statements** which preface each program; together, these job control statements constitute the **job control language** (JCL) of the machine. The executive control routines read these statements and check that the required resources are available; if they are not, an error is signalled and the program is not executed (*figure 4*).

Suppose you have a FORTRAN program to run and want the results printed. The FORTRAN statement for output is:

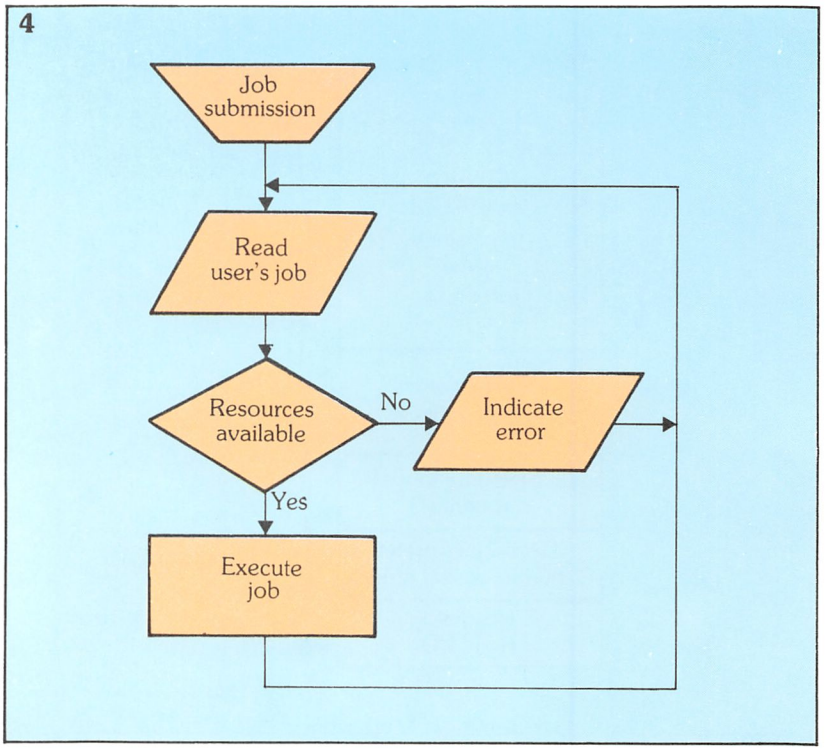
WRITE (7,100) X

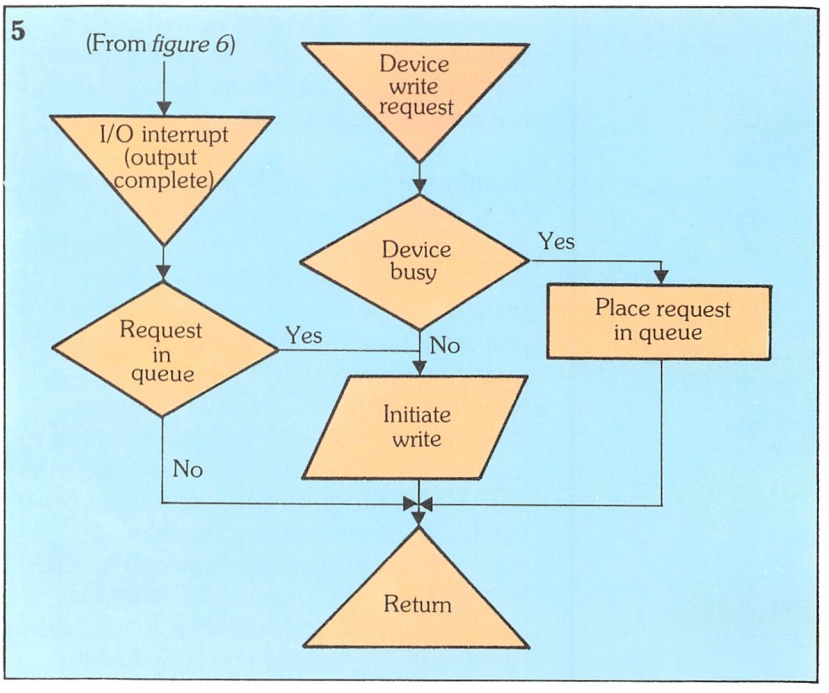
The number 100 refers to the instruction that specifies the output format (i.e. where and how the values of the variable X are to be printed); the 7 indicates that the output is to go to peripheral number 7, which in this case is a printer.

Time limits for programs can also be specified: these prevent the program from becoming stuck in a loop (by a 'bug' for example). The time limit is used by the executive to interrupt the job if it hasn't finished, and hand control to the next job. The executive also times each job, for accounting purposes.

The I/O control function handles the necessary code conversions, checks that devices are available and operational, and supervises I/O operations. The I/O processor is a hardware device used specifically for input/output operations, and carries out its functions alongside the CPU. In most personal computers, the I/O processor takes the form of an IC.

Suppose that a program has been loaded which instructs the computer to print out 100 words from a specific memory location. The CPU passes control to the I/O processor which begins printing. The I/O processor receives a signal from the peripheral when it has finished printing; if a second output request is then made for the printer, it knows that the first job has finished and initiates the second. If the first job had not finished, the second I/O print request is placed in a queue (*figure 5*).



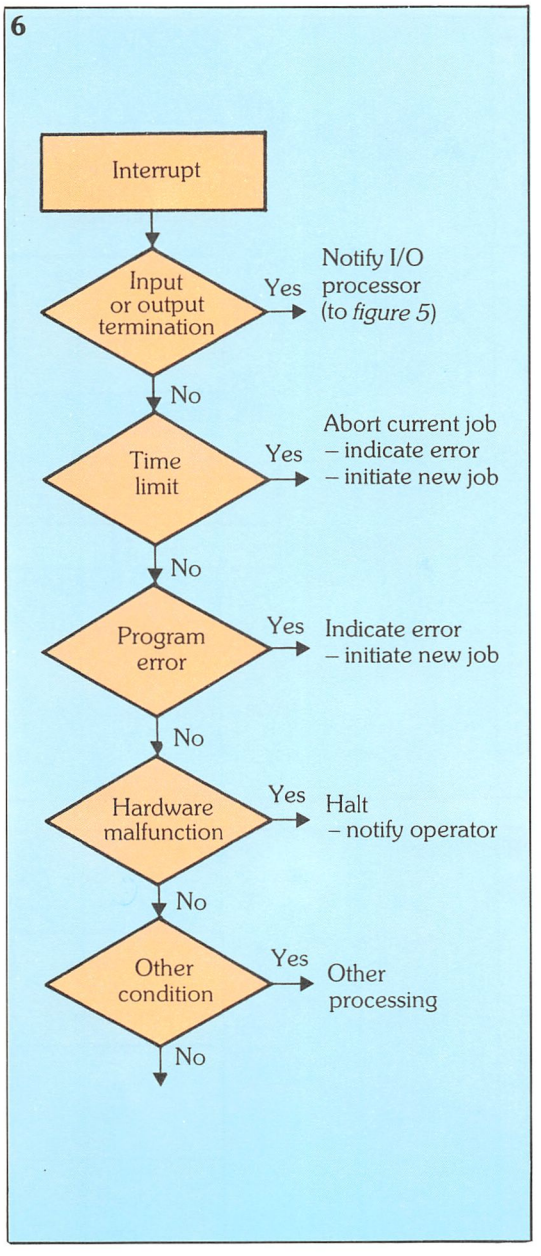


4. Flowchart for the resident monitor.

5. Flowchart for the I/O processor.

Interrupts

An efficient computer system needs a mechanism for breaking (or interrupting) the normal flow of instructions to the CPU when special conditions occur – this is achieved by using **interrupts** which are a function of the device hardware. Normal program execution can be halted and control



6. Using interrupts to interrupt the normal flow of instructions from the CPU when special conditions occur.

transferred to the executive which examines the conditions that caused the interrupt.

For example, an interrupt may signify the completion of an I/O operation by the I/O processor – the executive then hands over to I/O control as shown in *figures 5* and *6*. A programmer, for example, may try to divide a number by zero or perhaps try to reference a non-existent memory location (these errors are known as exception conditions): in either case an interrupt routine stops the program and an error message is issued. Also, if the preset time limit is exceeded, then the program execution is also interrupted.

Multiprogramming systems

During the mid-1960s a second step was taken towards increased efficiency of computer operation by the introduction of multiprogrammed operating systems.

Multiprogramming refers to a method of computer operation whereby two or more applications programs are run at a time. (Note: this does not infer simultaneous execution.) For example, suppose that you and a friend named Joe, each have a program to run on a batch processing system. Your program takes one minute to run, while Joe's takes one hour. If Joe's program is run first, you have to wait one hour and one minute before your results are ready (*figure 7*). On the other hand, if your program is run first, your result is ready just one minute after the program has been started.

Multiprogramming provides an alternative to this sequential batch processing. Suppose the operating system reads both of your programs into memory as shown in *figure 8*. The memory then contains the resident monitor routines, the machine instructions for your program and the machine instructions for Joe's program. With multiprogramming, the monitor allows both programs to share the CPU by alternately allotting an interval of processing time to each program: this time interval is known as a **time slice**. The CPU therefore swops between the two programs until one is completed. Your program now takes two minutes to run, instead of one, while Joe's program takes an hour and one minute to run. (This example assumes that the job – job switch occurs in zero time, although in practice this would not happen.)

The time required from submission to final results is often called job **turnaround**. In the batch processing example (*figure 7*), if Joe's program is run first the average turnaround time is $(60 + 61)/2 = 60.5$ minutes. With multiprogramming (*figure 8*) – again assuming that Joe's program is run first – the average turnaround time is $(61 + 2)/2 = 31.5$ minutes, which is a significant improvement on the batch system, or is it?

Figure 9 illustrates the total execution

time taken if three jobs of equal length are in the computer system. With batch processing (*figure 9a*), the average turnaround time is (1+2+3)/3 = 2 minutes. Using multiprogrammed processing, however (*figure 9b*), the average turnaround time has increased to 2.75 minutes. It is therefore evident that when all jobs require the same amount of CPU time, multiprogramming results in an increased average turnaround time for *all* jobs. The degree of improvement provided by multiprogramming therefore depends upon the mixture of CPU time requirements.

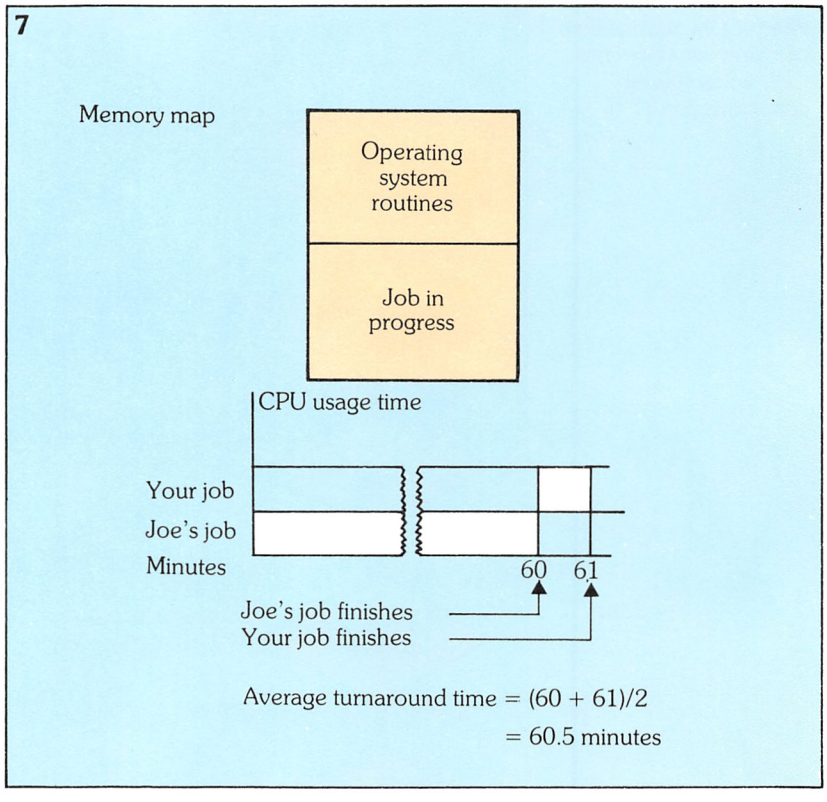
However, let's suppose that of the three jobs considered in *figure 9*, Job A needs to read data at the beginning of its execution, and that the program can't proceed until all the data has been read in. We'll assume that the processing takes a quarter of a minute and the reading takes three-quarters of a minute. Jobs B and C don't need to read in any data.

We can see from *figure 10* that the turnaround for Jobs B and C is decreased because they share the CPU equally while Job A is reading in data; jobs waiting for I/O are always in a 'hold' state. The average turnaround time is now 1.83 minutes. The CPU has been used for 100% of the time; if the three jobs were processed sequentially, the CPU would have been unused for three quarters of a minute out of the total 3 minutes of processing time. In this case, then, multiprogramming achieves a net improvement in turnaround and resource utilisation.

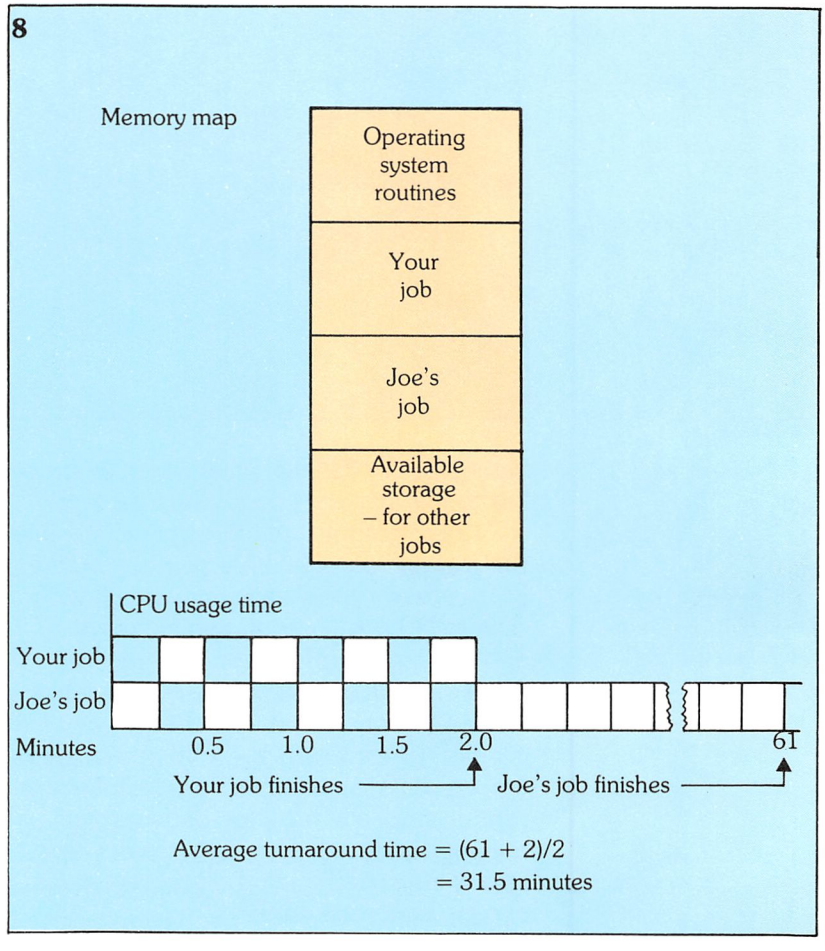
To sum-up, we can see that a major advantage of multiprogramming is more efficient CPU utilisation when lengthy I/O operations are carried out; when a job is not using the CPU because it is involved with I/O, the CPU can be assigned to another task. Multiprogramming also improves job turnaround when a stream of jobs with different execution times are run.

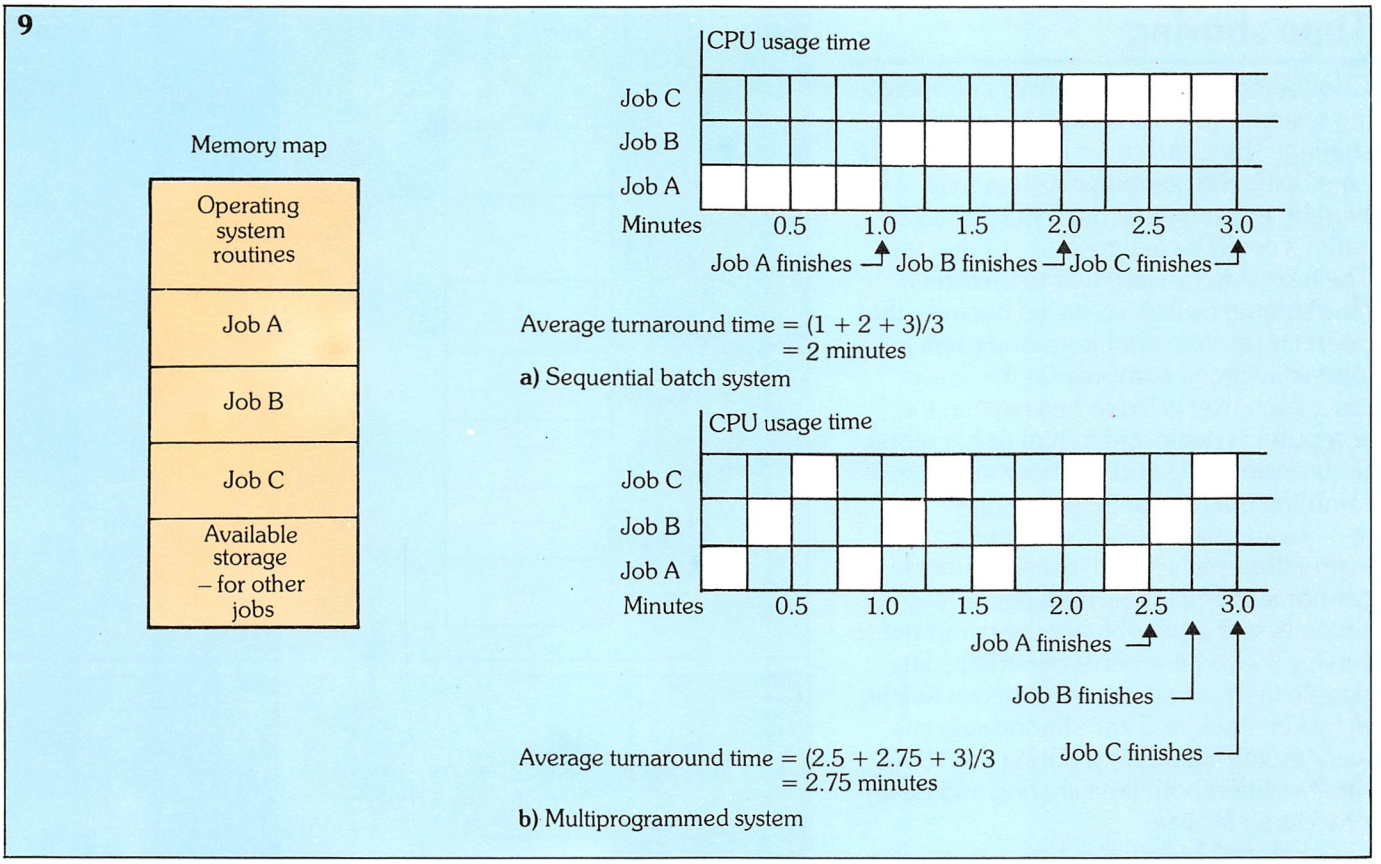
The development of operating systems capable of multiprogramming necessitated changes in computer hardware design. Such things as **monitor states** and **user states** were introduced: when the computer was in the monitor state, information could be read from any memory address; in the user state, only particular memory addresses could be accessed pre-

7



8



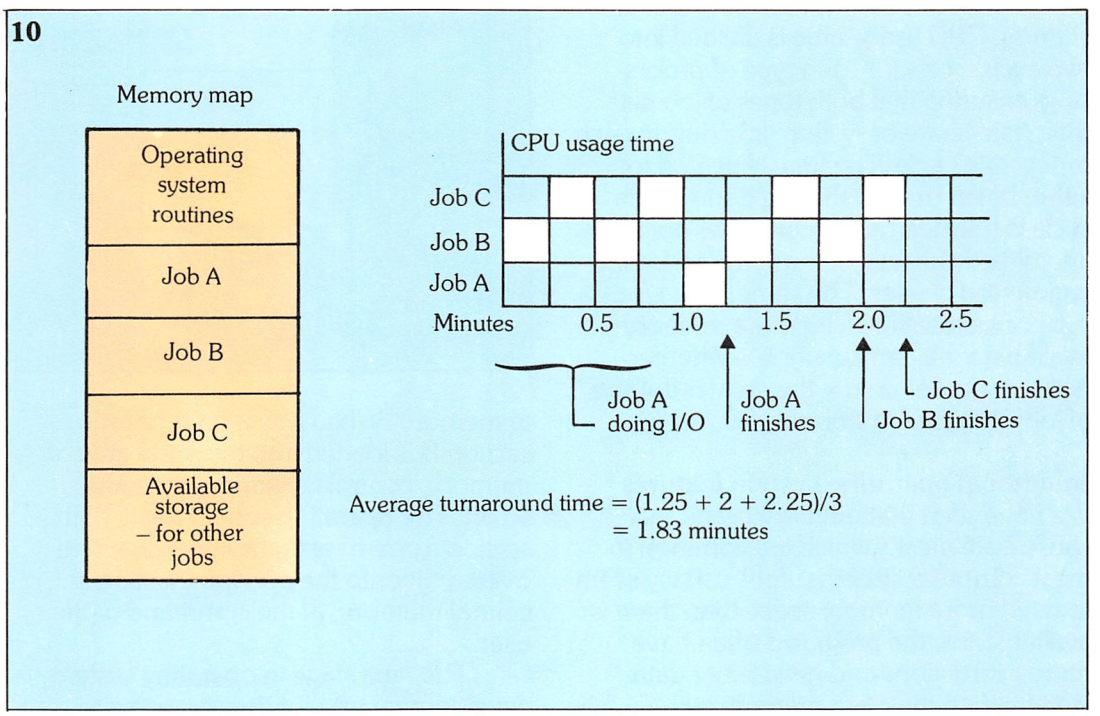


7. Two programs running sequentially on a batch processing system.

8. Multiprogramming will almost halve job turnaround by sharing the CPU time between programs.

9. Three jobs of equal length being processed on: **(a)** sequential batch system; and **(b)** multiprogrammed system.

10. How I/O affects average job turnaround in a multiprogrammed system.



venting one program from interfacing with another. An additional set of commands – **privileged instructions** – is utilised by the monitor state: one of the most important specifies memory boundaries. This is necessary because in order to process various programs efficiently, procedures were needed to relocate jobs to new memory areas.

Time sharing

Continuing this historical survey of operating systems takes us on to consider time sharing. The trend so far has been towards employing the computer system fully, thereby ensuring that the CPU is not idle when it could be assigned to another task. The next step forward was to provide a time sharing facility, so-called because the operating system enables many users to interact with the computer at the same time: each user is *led to believe* that the computer is dedicated to him or her alone. (In terms of CPU and memory, multiprogramming is a type of time sharing.)

Computers can execute millions of instructions per second; users, of course, cannot respond in less than one or two seconds so it is possible for the computer to service the requirements of perhaps fifty people in the time interval between keying in two characters. Time sharing systems were rapidly developed during the 1960s that provided both time sharing and batch processing facilities.

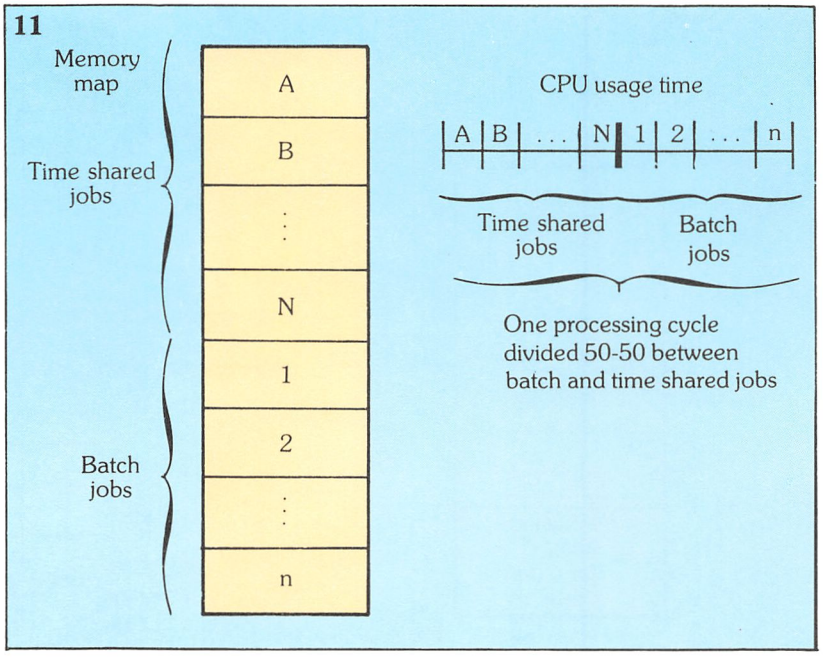
Figure 11 illustrates how a computer is used for both batch processing and time sharing. CPU usage time is divided into two cycles, one for each type of processing, ensuring that both types of job get their 'fair' share of system time during each processing cycle. The time allocated to either batch or time sharing during each cycle is selected by the computer operator and directly affects job turnaround for the various job classes. The same is true for memory allocation. The more memory available for a particular job – whether batch or time shared – the greater the size of jobs that can be stored.

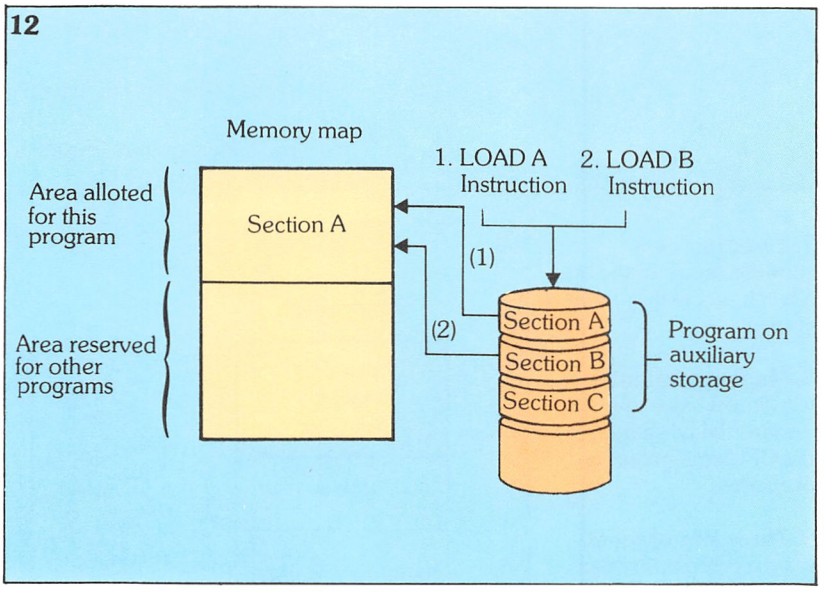
Additional operating system features

We have seen that memory usage can cause bottlenecks which are common to most computer systems. Jobs usually seem to need more memory space than there is available, i.e. the programs often have more instructions and need more data storage than the main memory can accommodate. This problem can be solved by the use of **overlays**.

Figure 12 illustrates a situation where programs are divided up into sections – each section being small enough to reside in memory. When a section A has run, section B is loaded into the same area of memory, control is handed over, and so on. The operating system takes each section from mass memory storage and overlays it onto the previous section in central memory, at the command of the user.

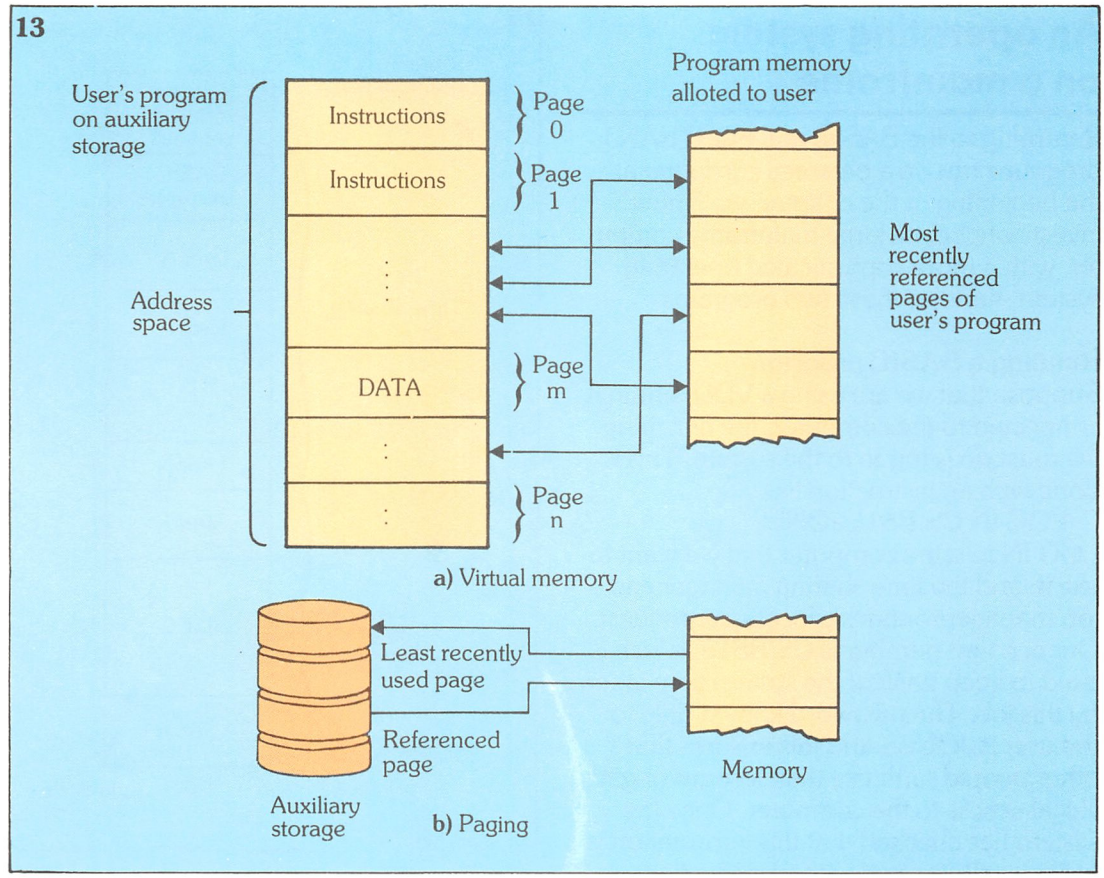
The next stage in operating system development was for the computer to automatically control the overlay process. The memory space (address space) required by a program is defined as being all addressable instructions and data – this can, of course, be greater than the memory



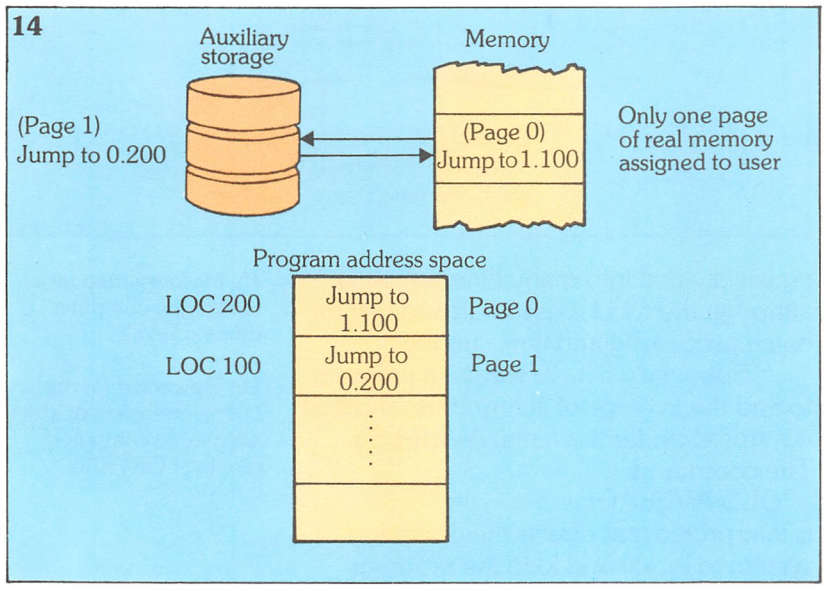


11. Using a computer for both batch processing and time sharing.

12. Programs can be divided up into sections and stored in auxiliary memory, then 'overlaid' into the same area of central memory when needed.



13. (a) Virtual memory which increases the amount of available memory space; and **(b)** the paging system.



14. Frequent page swapping, known as thrashing.

space available. The address space is divided into equal sized **pages** (*figure 13*) and loaded into the allotted memory space as needed; control can be passed to another job while this loading takes place. Once the referenced page is loaded, control is returned to that program. When a location is referenced that is not on the page held in memory, a **page fault** is said to have occurred. The memory available to a program can usually accommodate more than one page, so when a page fault occurs, the last used page is replaced by the one referenced. This is called the **least recently used strategy** or LRU.

One point to remember is that when a page is moved from mass memory to central memory, only a copy of it is transferred. All original pages are still held in mass memory. This kind of storage is termed **virtual memory**. Those pages which are replaced in main memory after they have been modified as part of the execution of the program (e.g. data tables) are loaded into a file on disk called the **swap area**. This area is interrogated before a page is loaded to see if a modified version is present.

Figure 14 illustrates **thrashing**, where frequent page swapping, which is due either to system overload or to the page size being either too small or too large, causes a high percentage of system resources to be devoted to paging, rather than program execution.

An operating system on a mainframe

Returning to the BASIC and FORTRAN programs run on a personal computer at the beginning of the chapter, we'll now investigate how a large mainframe computer, with a more sophisticated operating system, handles these two programs.

Running a BASIC program

Suppose that we are using a VDU terminal connected to the computer, the first thing we must do is **log in** to the system. This is done with an instruction like:

LOG IN CS.B815 JONES

LOG IN tells the computer that we want to use it, and the time sharing manager and job manager routines process our request. Our account number is CS.B815, which is used to keep track of the system time taken by this job. The password known only to the user is JONES and this ensures that other people can't use that account or gain illegal access to the computer. Once the system has checked that this information is valid, it will respond with a verification procedure such as:

JOB 542 LOGGED IN ASSIGNED
TTY 14

which simply indicates the number assigned to our job, and that the terminal we are using is number 14. The operating system then reserves some memory space and CPU time to run the job.

We now need to ask to use the BASIC interpreter, which is achieved by typing in:

BASIC

The system's memory map is shown in *figure 15*, and we can see that there are several other users: some using BASIC on time share and others using batch processing.

The BASIC interpreter has some of its own system commands for writing or executing BASIC programs which it must run through the operating system. The operating system first passes control to the BASIC interpreter where each BASIC instruction is converted to machine language and then executed. Control is also passed to each batch program that is sharing memory. Each unit of processing time is divided into many different parts, although the CPU is used half and half for batch processing and time sharing.

Suppose we have written a program to find the average of N numbers, and that it is stored under the name AVERAGE. The command:

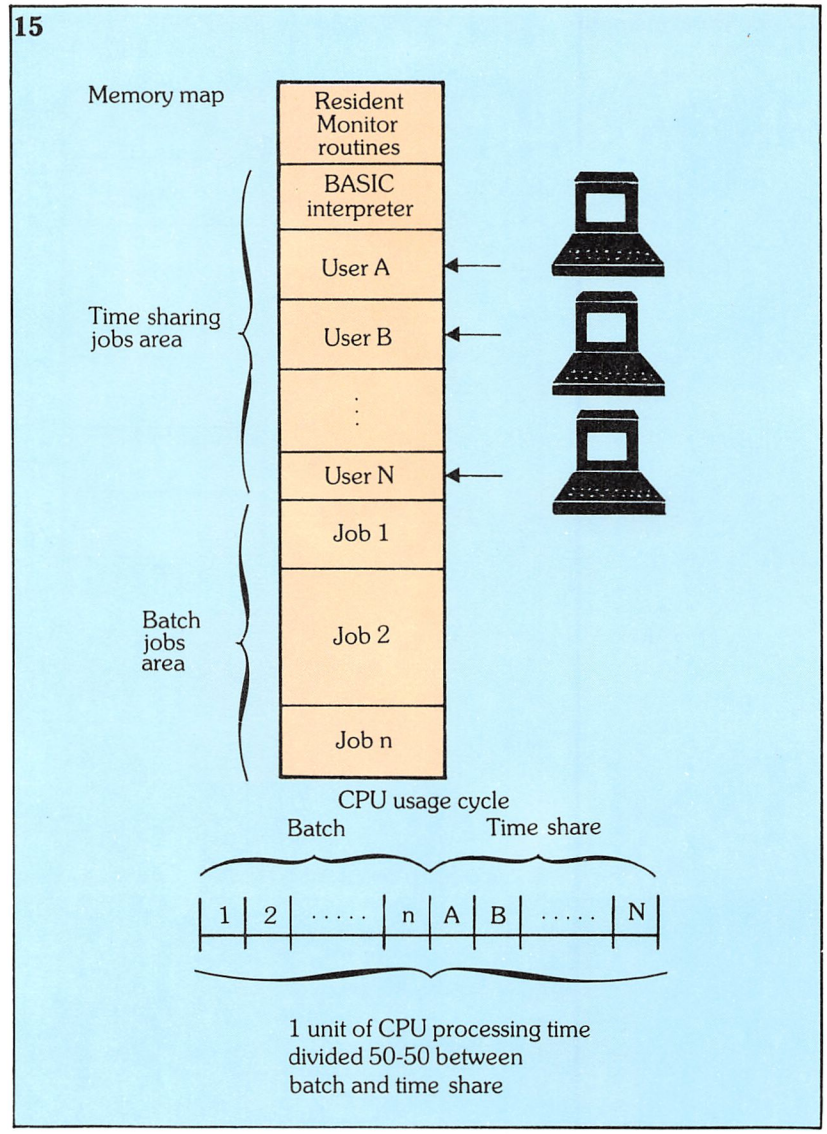
OLD AVERAGE

is interpreted and directs the operating system to locate and load the program called AVERAGE. (The prefix OLD, tells the system that it has been stored previously). While the program is being loaded into the memory area assigned to it, control is passed to other jobs. Once loaded, we type:

RUN

which (once control has been passed back to our program) instructs the BASIC inter-

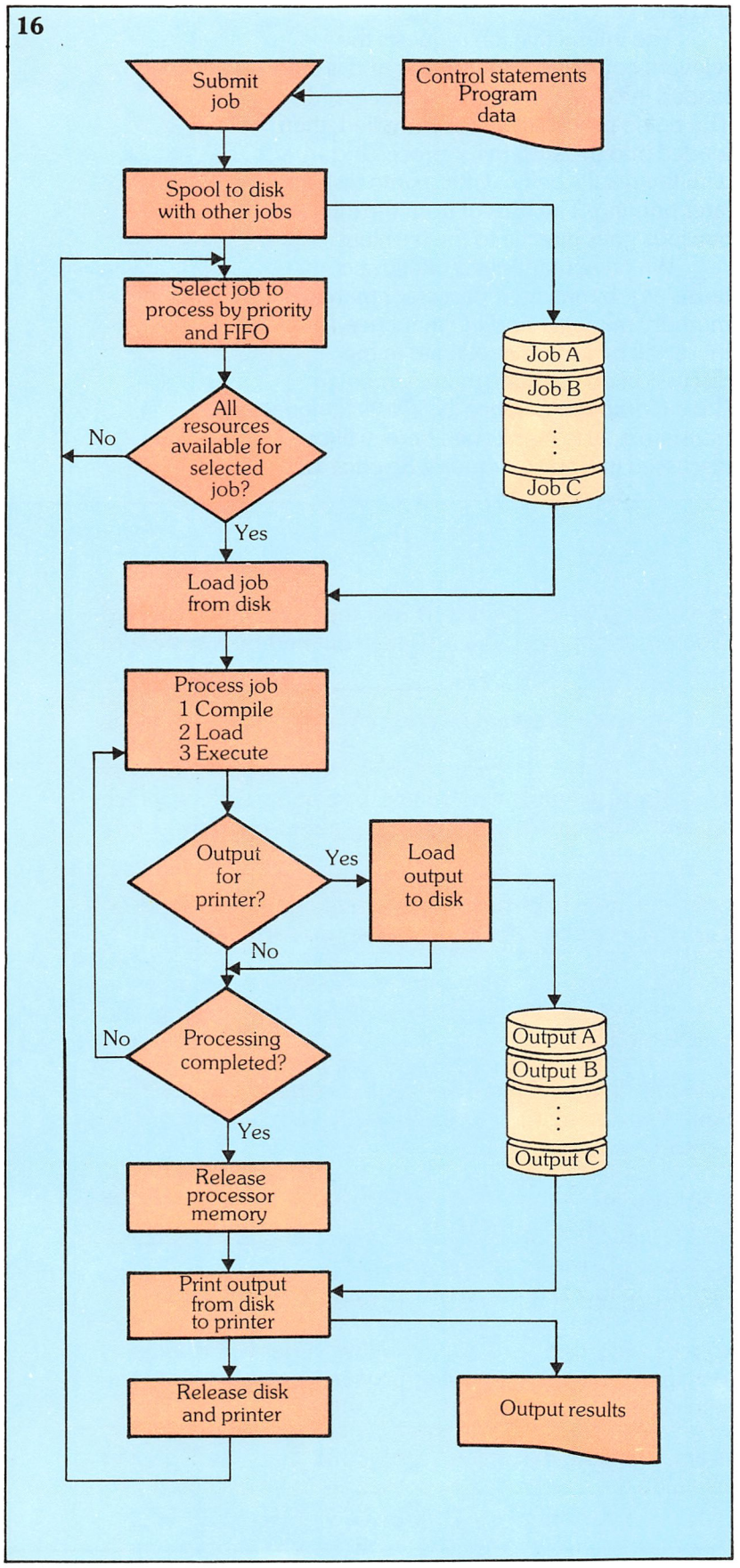
15



15. Memory map for a mainframe computer running BASIC.

16. Flowchart for the operating system of a mainframe computer running FORTRAN.

16



preter to begin executing the BASIC instructions in our program. As translation is through an *interpreter* and not a compiler, the operating system executes each statement after its translation when it is our turn in the time sharing cycle.

The operating system is also responsible for reading and writing to the terminal device. All I/O is handled by the I/O manager. The user's BASIC program requests output, but the operating system actually writes it.

Running a FORTRAN program

FORTRAN doesn't have as many system functions as BASIC and, generally, operations are performed through the operating system. Each job comprises job control statements, the program and data and is broken up into tasks to be executed.

If the job is to be batch processed, it is read in with any others and loaded onto disk (*figure 16*). As this happens its entry time, priority and resource requirements are noted. A set of routines known as the job manager examines the priority and entry time. Suppose that our job has the highest priority – the job manager examines the resource needs, which include memory and the I/O specified by the job control statements. The result of this is that a job manager subroutine requests the memory manager routines for memory space.

Once the resource needs are met, our job's first step or task is initiated – which will mean loading the FORTRAN compiler if this has not been done. When our job has been compiled, the job manager requests that a loader program loads it into the system. Control is then passed to our job and its execution begun.

When our program requests output to the printer, it is not printed directly, but written onto disk to be printed out at a later stage. This ensures that CPU time is not wasted by waiting for a (relatively) slow printer to operate.

On the other hand, some mainframes allow interactive execution of FORTRAN programs. In this case, an operating system program called an **editor** may be used, which allows the creation and interactive editing of FORTRAN files on disk. The disk files can then be compiled with the use of a

system command like:

FORTRAN PROG

after which the required program can be run.

We can see that many system routines are used to process our program. These initiate the job, monitor its execution and stay with it until it is complete. Each job in the system – and there can be many – will likewise have a manager controlling and monitoring its process. Some operating system modules are shared between the interactive and batch processing applications; both make requests to the job manager, but they are handled differently – interactive jobs are processed immediately while batch jobs are first stored on disk.

The interactive job requests the relevant compiler or interpreter and is then loaded into the system, to be processed. The batch processed job is compiled, then loaded into the system for processing – with the results being written onto disk for later printing. The output from the interactive jobs goes directly to the terminal.

We have mentioned briefly such terms as job manager, processor manager, memory manager and I/O manager – these will be further explained in the discussion on resource management in *Basic Computer Science 11*. (Note: language translators, also mentioned here, will be discussed in *Basic Computer Science 12*.,

Glossary

batch processing	form of data processing where programs and their data are executed individually. The order of processing is usually defined by a system of priorities
compiler	program used to translate programs written in a high level language into machine code
editor	collection of programs within a computer's operating system which allows loading and subsequent editing of programs on disk before execution
file	this is a collection of user information (programs, data etc.) held on back-up storage which can be referred to by a single name
housekeeping routines	software that helps organise the use of available resources – particularly with respect to storage management, e.g. opening and closing files, performing I/O etc.
interpreter	program used to translate high level language programs into machine code, statement by statement. Typically used with an interactive language like BASIC
multiprogramming	form of data processing that enables two or more programs to be executed at a time in a computer system
operating system	set of programs provided by a computer manufacturer, that control and manage the execution of the user's programs
time sharing	form of data processing that uses an operating system that enables many users to interact with the computer, which appears to be dedicated to each user
virtual memory	operating system function that allows programs that need greater storage capacity than the main memory possesses, to be executed

ELECTRICAL TECHNOLOGY

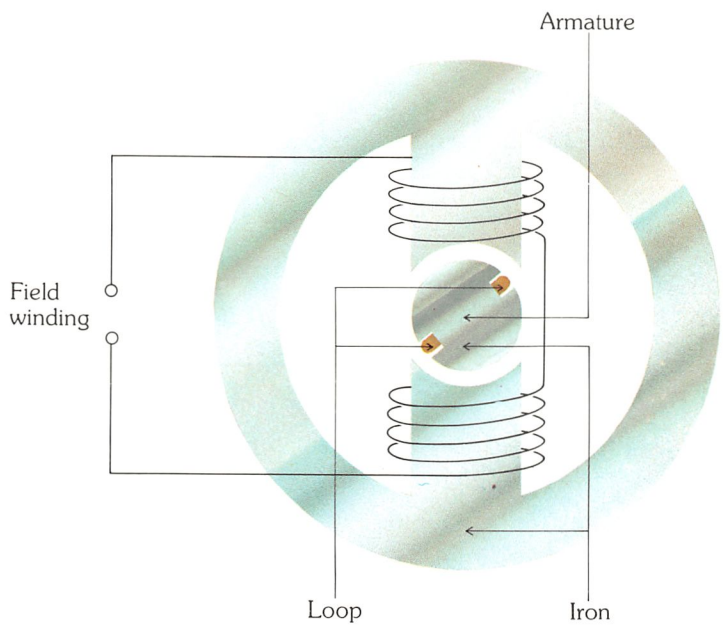
Conversion of energy-1

One of the principle uses of electricity is to provide power for the operation of electrical equipment – TVs, radios, cookers, lights etc. This power may be obtained by using some primary source of energy, such as the energy contained in fossil fuels or the nuclei of atoms, radiation from the sun, or heat from the interior of the earth, to, say, drive an electrical generator. It is to this subject of electrical generation and the various methods involved that we now turn.

Electrical power generation

In the previous *Basic Theory Refresher* we saw how an electrical voltage may be generated by the rotation of a loop of wire in a magnetic field. Although this concept is hardly suitable for the production of large quantities of power (remember, the voltage generated fluctuated continuously between positive and negative values, one cycle of oscillation occurring for each revolution of the coil) the principle, however, forms the basis of the **alternator**, or AC generator.

1



An alternator – constructed in the form of an electromagnet wound with a coil supplied with a steady current and mounted on an armature.

The performance of this simple alternator can be improved by making a few modifications. The first of these modifications relies on the fact that the generated voltage, and hence the power available, is dependent upon the magnitude of the flux in the air gap in which the loop rotates. A permanent magnet will give a rather low flux and is relatively easily demagnetised, we shall therefore construct the alternator in the form of an electromagnet, wound with a coil supplied with a steady current.

The magnetic flux produced by an electromagnet is very much greater when the flux path is filled with a ferromagnetic material. The second improvement we can therefore make is to mount the moving loop on an iron core, called the **armature**, as shown in *figure 1*.

A third improvement is to replace the single loop of wire with a coil of many turns on the armature. Finally, we see from *figure 1* that only a very small part of the circumference of the armature has been used and so the last improvement is to increase the generated voltage by placing a number of coils in adjacent slots in one region of the armature, and connecting those coils in series to give the total output voltage.

Eddy currents

We have already seen that any piece of conducting material which is in motion in a magnetic field will have an EMF induced in it. In particular, if an iron core is used as the magnetic armature in an electrical machine, an EMF will be induced in it which, in turn, drives a current through the material causing circulating currents (or eddy currents) to be set up within the iron. These currents will dissipate energy, in the form of heat in the core, constituting a loss of energy in the machine; in other words, the output electrical power will be less than the input mechanical power. These eddy current losses must be minimised in order to make the machine as efficient as possible.

The obvious method of reduction is to increase the resistance of the eddy current path to as high a value as possible, either by making the resistivity of the iron large or by making the current paths long and narrow. The first method is relatively simple; the second is a little more involved and is achieved by constructing the armature from a number of very thin sheets (**laminations**) of iron, each one insulated from its neighbour and bolted together so that they are all perpendicular to the axis of rotation. The eddy currents are thus forced to flow through a very narrow path and the energy losses are consequently reduced.

Another form of eddy current loss is associated with the hysteresis loop which we have already met. Consider a piece of magnetic material initially magnetised in one direction; now let the direction of magnetisation be reversed to its maximum negative value and

then reversed again, returning finally to its original value, thus taking the material through one complete cycle of magnetisation. Energy will be lost during this cycle, the amount of which is proportional to the area of the hysteresis loop. Looking at the simple alternator shown in *figure 1*, we see that as the armature undergoes a reversal of magnetism in a hysteresis loop, the iron experiences a **hysteresis loss** as well as the eddy current loss. These losses, known as the **iron losses**, reduce the efficiency of the alternator.

A further factor to be considered is the power loss associated with the current flowing through the resistance of the coils. This also further reduces efficiency.

In most practical alternators, the machine as we have described it is turned inside out as shown in *figure 2*. The electromagnet which creates the magnetic field (called the **rotor**) rotates inside the **stator**, a stationary piece of iron which holds the coils in which the EMF is generated. In order that electricity reaches the rotor, the ends of its winding are connected to two annular rings mounted on the central shaft: two pieces of carbon (**brushes**) press against these and conduct the current.

Direct voltage generator

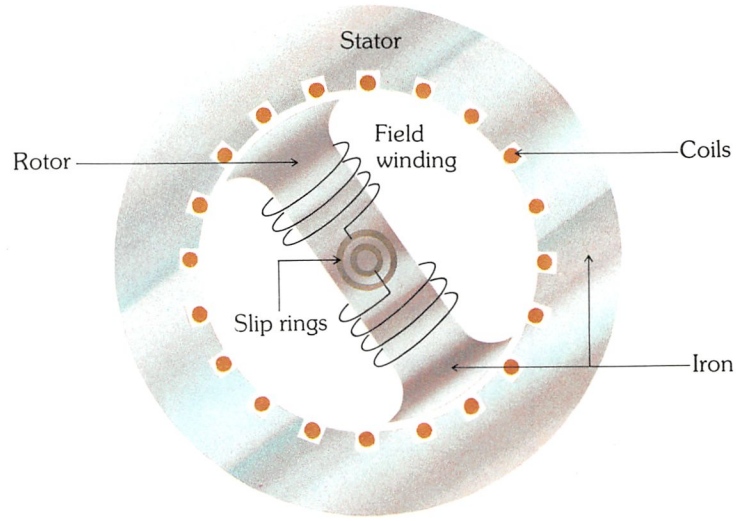
The alternator is an AC generator and is used for the majority of applications – the National Grid mains electrical system, for example, is based on AC electricity and uses alternators for generation. Some applications though, for example car electrical systems, require DC electricity and a few modifications to the primitive alternator of *figure 1* can provide this.

Looking at *figure 1* and also at the shape of the voltage generated shown in *figure 6b* from the previous *Basic Theory Refresher*, we can see that if we change the direction of the voltage every time the loop of wire passes the horizontal position, we shall have a voltage whose magnitude rises and falls but which is always in the same direction. To do this we connect the ends of the coil to the two halves of an annular ring (known as a **commutator**) as in *figure 3*. Two fixed brushes are then pressed against this commutator and, as the armature rotates, the voltage at the terminals attached to the brushes is always in one direction.

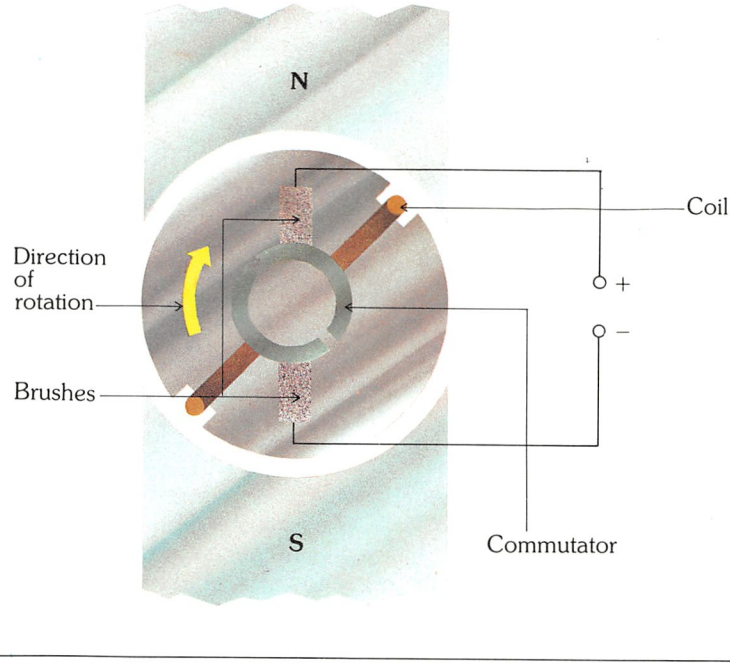
This arrangement can be improved by using a large number of coils, placed in a series of adjacent slots around the armature. Each coil can be connected to two segments of a multisegment commutator resulting in the output voltage from each coil rising to a maximum at a different angle in the rotation of the armature. In this way, when the voltage from one coil is zero, that from another coil may be at maximum. If all these coils are connected together, the output voltage from the brushes will not only be unidirectional but will also have a magnitude which is more nearly constant.

DC generators are often grouped according to the way in which the magnetic field is produced. The field windings, for example, may be fed from an independent supply of voltage giving rise to the **separately excited** generator. However, as the generator is itself

2



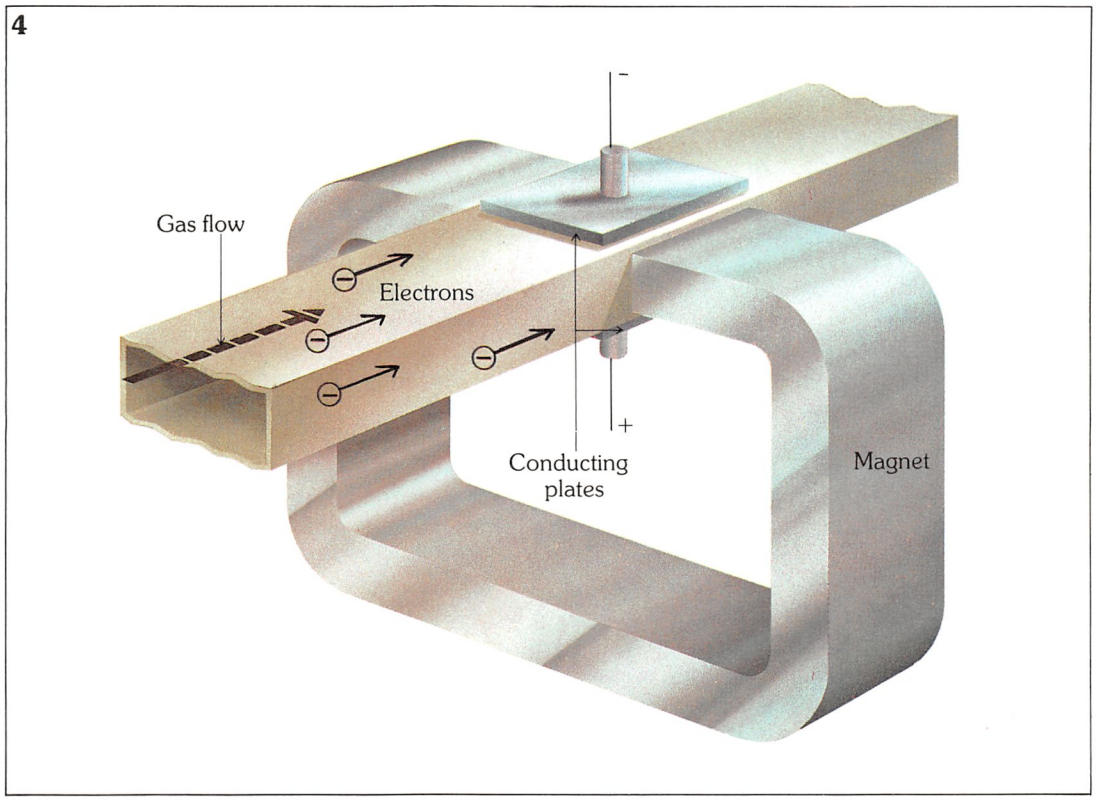
3



2. Practical alternator – figure 1 turned inside out.

3. An alternator used for DC supply.

4. Magnetohydrodynamic generation.



producing a direct voltage, we may use it to provide the supply to its own field windings since only a small current is required; such machines are known as **shunt wound** generators since the field winding is in parallel, i.e. shunts across the output terminals. Another alternative is to connect the field windings in series with the load giving a **series wound** machine.

DC generators are normally only used in specialised applications, as it is simpler and cheaper to have an alternating voltage from an alternator and rectify it using semiconductor devices.

Magnetohydrodynamic generation (MHD)

A recently developed method of electrical power generation passes a stream of electrically charged gas through a tube placed between the poles of an electromagnet, as shown in *figure 4*. As a result, a voltage is developed between the opposite walls of the tube. This technique has the great advantage that no rotating pieces of machinery are involved. However, its efficiency is lower than that of the generators we have just seen.

Solar cells

In countries where the sun shines for a large part of the day it is possible to use large arrays of photoelectric cells to generate electricity directly from sunlight. Again, efficiency is low but the cost of fuel is zero. Such power cells also provide the electrical supplies for satellites.

Fuel cells

The most common device for generating electricity is the battery which we use in electric torches, transistor radios, etc. The chemical energy of the electrolyte in a battery is converted directly into electrical energy. A more powerful version of this is the fuel cell, in which two chemicals, usually oxygen and hydrogen gases, are combined to form water and the extra energy from this chemical reaction is released as electricity.

Fuel cells directly converting the energy of hydrocarbons can give efficiences around double that obtained by burning the fuel to provide the mechanical energy to turn an alternator. They also have the great advantage of silent operation. However, the usage of batteries or fuel cells on a large scale would necessitate the continuous replacement of the basic chemicals. □

BASIC THEORY REFRESHER

ELECTRICAL TECHNOLOGY

Conversion of energy-2

In the previous *Basic Theory Refresher* we looked closely at various methods of power generation. The converse process to this energy conversion is the utilisation of the electrical energy so generated in factories, homes and offices.

Heat and light

Two of the most obvious energy converters are the electric light bulb (converting electrical energy to light energy) and the electric heater (converting electrical energy to heat energy); both of these work by effectively connecting a resistor across the electricity supply.

Eddy current heating

The eddy currents observed in the previous *Basic Theory Refresher* were undesirable because of the efficiency losses they caused in electricity generators. However, there is at least one application where eddy currents may be put to commercial advantage – when heating large quantities of metal, say in a metal foundry. A crucible containing raw metal ingots is placed inside a coil of wire and a large alternating current of a fairly high frequency is passed through it, causing an EMF to be generated in the metal. The resistance is quite high as the ingots only make contact with each other at isolated points – a large amount of power is therefore dissipated which melts the metal. This method of **induction heating** is preferable to combustion methods because no contaminating waste products are formed.

Electric motors

The conversion of electrical energy into mechanical energy to drive motors is one of the most common uses of electrical power. Motors use the basic principle that a force is experienced when a current is passed through a conductor placed in an electric field.

Synchronous motor

An alternator can be used to construct a motor by passing an alternating current through the alternator's winding – its rotor will turn and this can be then used to drive a shaft connected to some piece of machinery. Such a device is termed a **synchronous** motor and the cross-section of a simple one is shown in *figure 1*.

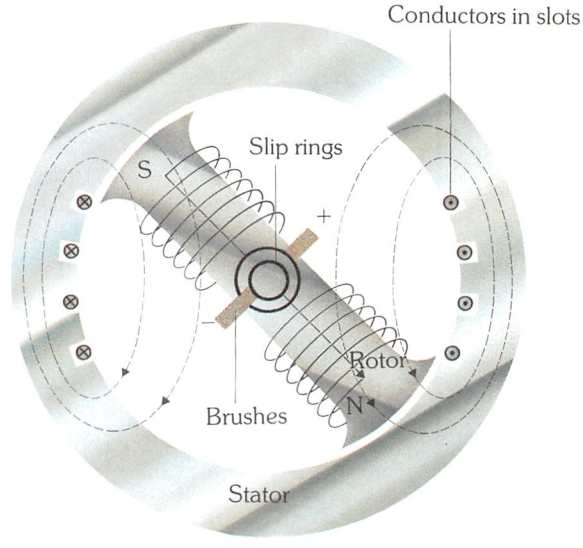
A direct voltage is connected to the brushes which causes a current to flow through the rotor winding in the direction shown. Using the corkscrew rule, it can be seen that this sets up a flux such that the armature becomes magnetised as shown. Now assume an alternating current is passed through the stator which, at the moment we are considering, is going down into the paper on the left hand side (as shown by the crosses) and coming up out of the paper on the right hand side (shown by the dots); a magnetic field is produced as shown by the dotted lines.

Recalling that an N pole will tend to move in the direction of the lines of force and an S pole in the opposite direction, we see that the N end of the armature is pulled downwards and the S end pushed upwards. It is this force which causes the armature to rotate clockwise until it reaches the vertical position.

At this point, the alternating current which is being supplied to the stator changes direction, and the magnetic field now points upwards: this continues to force the armature to rotate clockwise for another revolution. Thus, if the alternating supply reverses direction at each half revolution of the shaft, the armature continues to rotate – indefinitely.

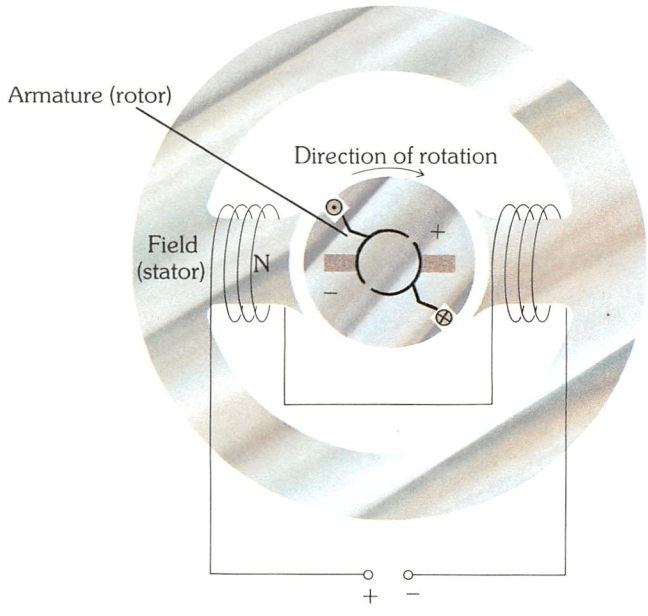
In fact, this occurs automatically because the speed of rotation adjusts itself so that one revolution exactly corresponds to one cycle of the alternating supply. We can therefore see

1

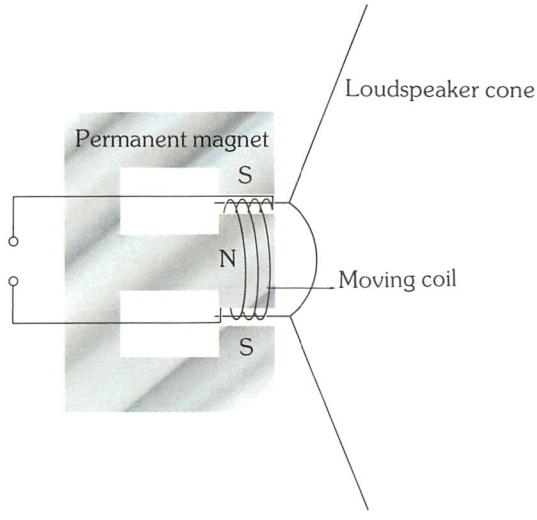


1. Cross-section of a simple synchronous motor.

2



3



2. A DC motor.

3. A moving coil loudspeaker.

that a synchronous motor, as its name suggests, is **synchronised** with the frequency of the electricity supply (which in the U.K. is 50 Hz). This property allows motors, such as this one, to be used in applications such as electric clocks where the speed is critical: in this case the rotor is a permanent magnet, rather than an electromagnet.

DC motors

The same principle is used in the construction of motors driven by a unidirectional voltage. Normally, however, DC motors have the field winding on the *stationary* part of the machine and the conductors on the *moving part*, as in *figure 2*. Current from the electricity supply flows through the field winding, magnetising the pole faces as shown. With the armature in the position shown, current through its coil (via the brushes and commutator) sets up a magnetic field, which rotates it in a clockwise direction. When the loop of conductors on the armature is vertical, the commutator reverses the current direction and so the armature continues to rotate clockwise. We can see that the action of the synchronous motor and the DC motor are identical except that in the synchronous motor the alternating current *automatically* changes the direction of flux, whereas in the DC motor the direction of current is changed *by the commutator* as the motor rotates. Consequently, the speed of a DC motor, unlike that of a synchronous motor, is not fixed.

Induction motor

Another type of motor which runs from an alternating supply is the **induction motor**. Unlike the synchronous motor, only the stator of an induction motor is supplied with an alternating current – the windings on the rotor have no supply connected to them. When the rotor is stationary, the magnetic flux in the stator links with the rotor coils generating an EMF. This EMF drives a current through the rotor windings producing a magnetic field. The magnetic fields of the stator and rotor repel each other causing the rotor to rotate. This rotation continues since the flux will change its direction every half rotation of the rotor.

Loudspeakers

A number of other devices convert electrical power into other forms. One example is a **loudspeaker**.

A diagram of a **moving coil loudspeaker** is shown in *figure 3*. When current flows through the coil in the direction shown, a reaction between the field and the permanent magnet's field exerts a force on the cone, pushing it outwards (i.e. to the right); when the current reverses direction, the force acts in the opposite direction. Thus, if an alternating current passes through the coil, the loudspeaker cone moves in and out setting up sound vibrations in the air. □

15

DIGITAL ELECTRONICS

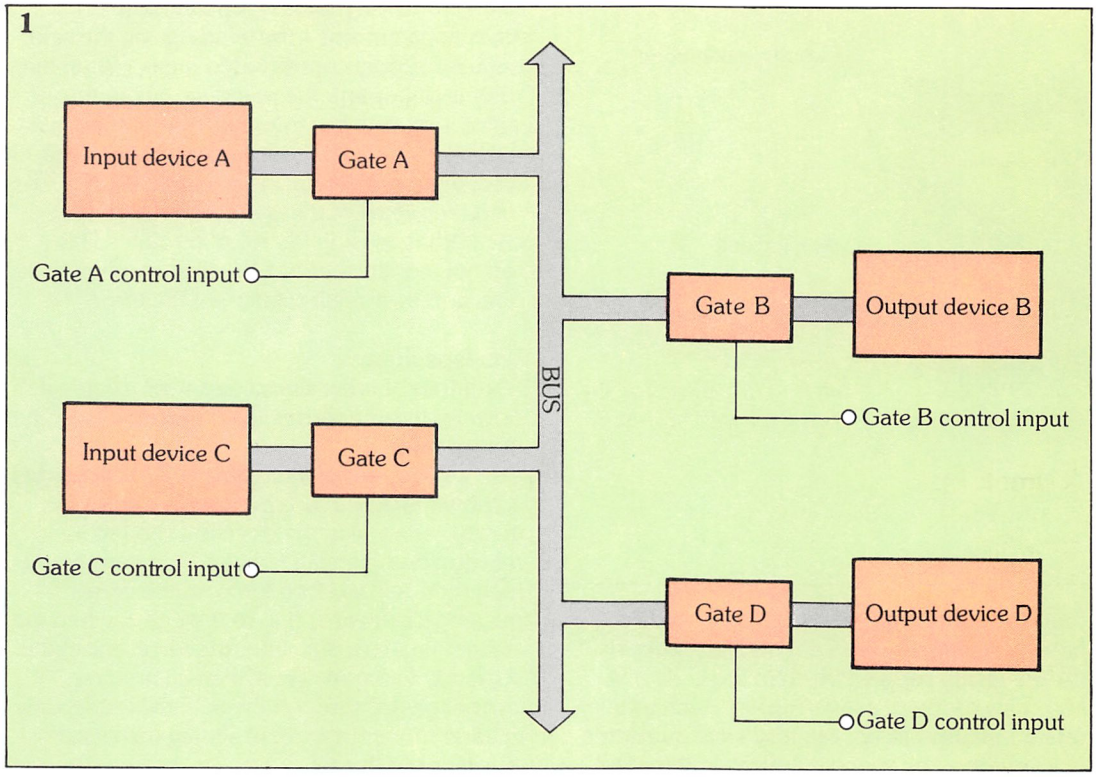
Busses

Three-state and open collector output stages

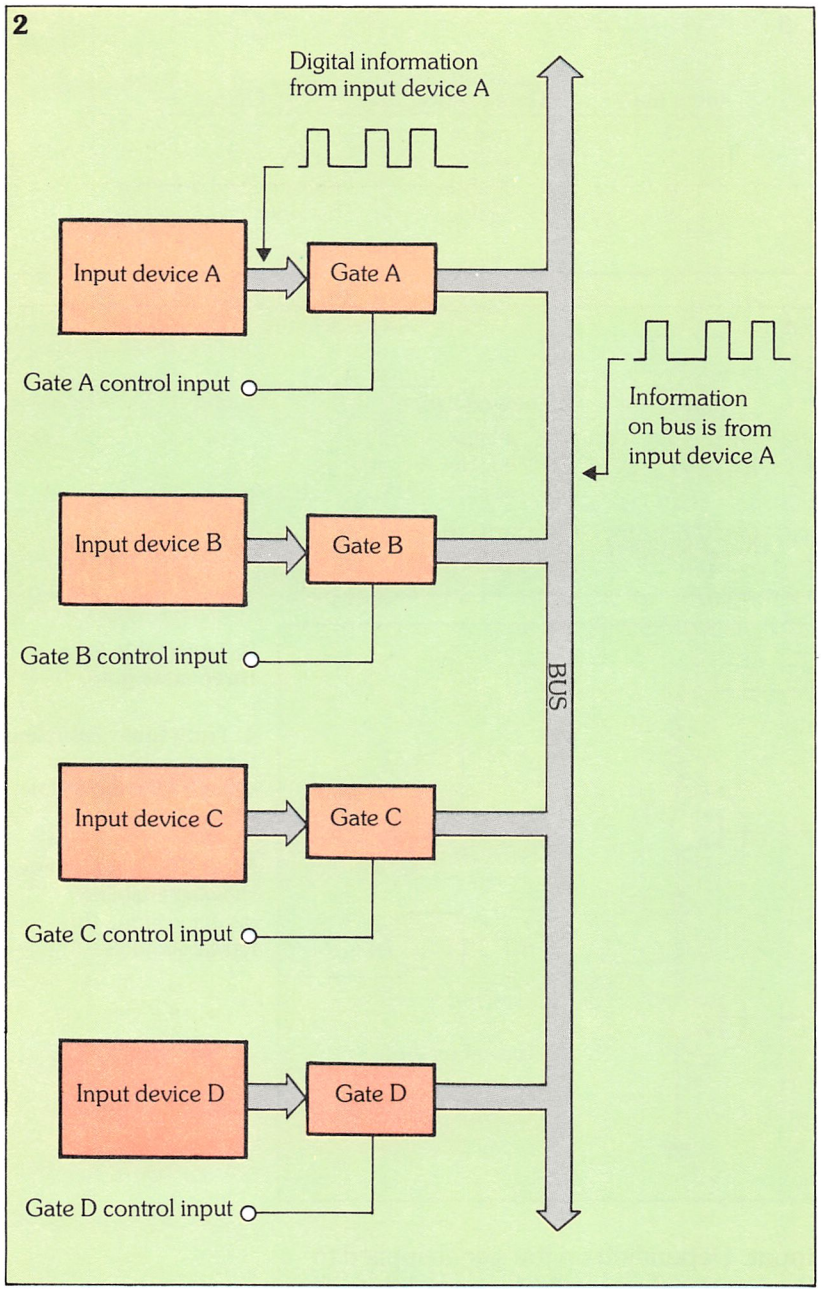
In *Basic Computer Science 2*, we saw that the various codes representing information in a digital system are transferred between devices over groups of wires known as **busses**. For example, the CPU of a computer system may communicate with one of many I/O devices (say, a VDU) over the system's data bus. Similarly, an I/O device may use a bus to communicate with one of several memory devices. Busses, then, form a convenient method of reducing the number of interconnections in a digital system (think of the great mass of separate wires that would be needed to connect each part of a computer to all the other parts, if busses didn't exist).

Bus systems joining many digital devices are commonplace in digital electronic systems. We have already seen how computer systems rely on the use of three main internal busses to transfer data, control, and memory address information between the separate digital devices. Another important use of bus systems is in modern scientific laboratories – many measuring instruments may be connected by a bus running around the laboratory to a central controller (generally a computer), which is, in turn, connected to output devices. In this way, experiments or processes may be accurately recorded and controlled with a minimum of active human involvement.

However, the use of busses to connect the parts of a digital system creates its own problems. The fact that many different devices may be simultaneously wired to a single bus means that the data flow must be disciplined in some way so as to



1. A simple bus system.



2. A bus system where four input devices (A, B, C and D) are joined by four gates.

allow only one communication at a time and only those devices which are creating the data are connected at that time.

A simple bus

Figure 1 illustrates a simple bus system. Two input devices (A and C) are to transmit information on the bus, and two output devices (B and D) are to receive it. Each input devices is joined to the bus by a gate, which can effectively connect or disconnect the device depending on a control input. Similarly, each output device is joined by a gate, which can connect or disconnect the device to or from the bus. The most important points which concern the operation of this system are:
1) if gates A and C are disabled, no information is present on the bus;
2) if gates B and D are disabled, no information is received from the bus;
3) gates A and C cannot be enabled simultaneously. At least one must be disabled at any point in time;
4) gates B and D can both be enabled simultaneously.

There are six possible ways in which this system can be used to transfer information between input and output devices:
1) from A to B;
2) from A to D;
3) from C to B;
4) from C to D;
5) from A to B and D;
6) from C to B and D.

The problem of interconnection is illustrated in *figure 2*. Here, four input devices (A, B, C and D) are joined to a bus by means of four gates. These gates are not formed from typical TTL or CMOS IC gates because they provide no facility for *disconnecting* devices which are not required to transmit information. In the example of *figure 2*, we can see that information in the form of a digital signal is present at the output of input device A. Gate A is activated and so the information present on the bus is the digital signal from input device A. We can draw an analogy with this system to a telephone circuit in which many telephones are connected – but only one telephone is being used to transmit. Electrical signals corresponding to the human voice will be present on the telephone line.

Now, if we imagine several telephones being used at the same time on the same telephone line, we can see that the resulting electrical signal will be a garbled mess which cannot be understood. Obviously, only one telephone can be used to transmit at any one time; all the others must be disconnected.

The bus systems of *figures 1* and *2* suffer from the same problem as in the telephone analogy – if more than one input device is connected to the bus at any one time, the resulting information on the

bus will be garbled, and no output device will be able to understand it. As in the telephone analogy, all input devices except one must be disconnected from the bus.

Three-state outputs

Ordinary logic gates cannot provide this disconnection facility – the output of such a gate must be one of the two logic states, 0 or 1. However, some special gates exist in both TTL and CMOS logic series' which can effectively disconnect input devices from a bus depending on control signals to the gates.

Figure 3 shows the truth table of such a **three-state gate**. We can see that when the gate is enabled by the control signal, whatever logic input is applied to the gate will be present at its output and connected to the bus. However, when the gate is disabled by the control signal the output is disconnected from the bus. This third state of the three-state gate is created by using the output stage of the gate to present a high impedance between gate output and input. Summarising, as shown in the truth table of *figure 4*, we can say that three possible output states exist:

1) a logic state 0;
2) a logic state 1;
3) a high impedance output state which effectively disconnects the input device from the bus.

All three-state gates have a control input known as the **enabling/disabling input**. Depending on the signal applied to this input, a three-state device behaves like a normal TTL or CMOS gate when it is enabled, or like an effective open circuit when it is disabled.

3

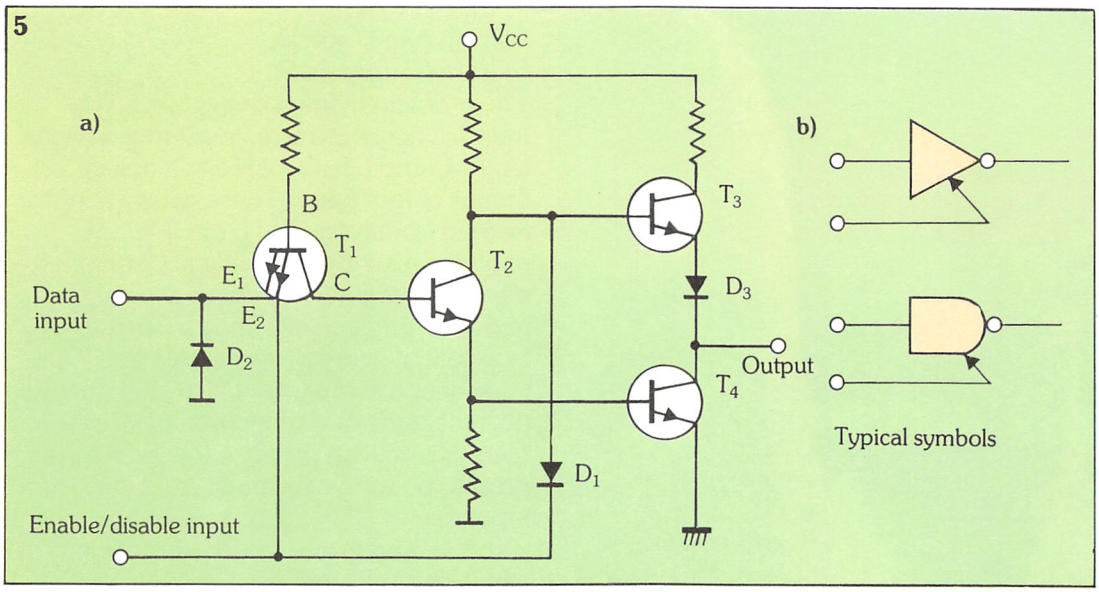
Input data	Gate control	Output
0	enable	0
1	enable	1
0	disable	disconnected from bus
1	disable	disconnected from bus

4

Input data	Gate control	Output
0	enable	0
1	enable	1
X	disable	high impedance

(X = don't care)

5



3. Truth table for a three-state gate.

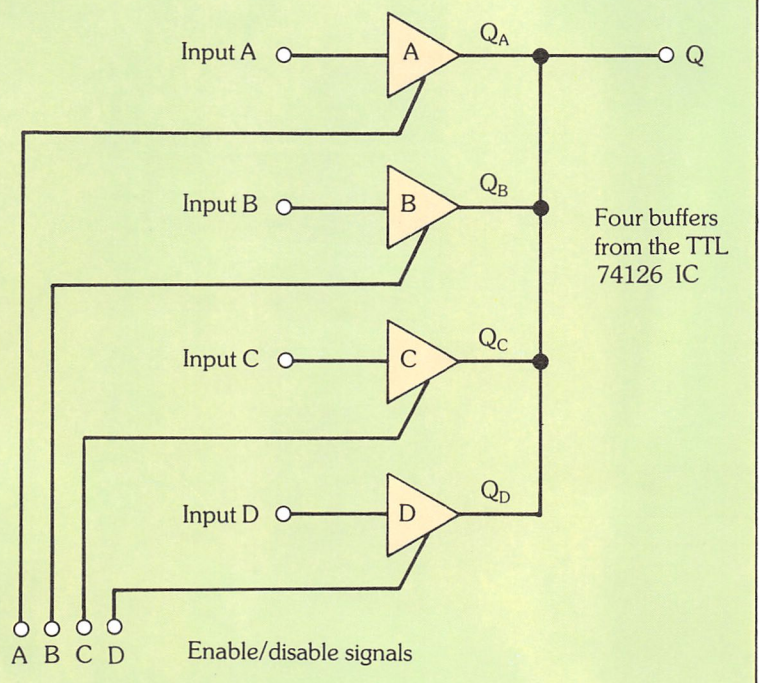
4. Truth table illustrating three possible output states for the three-state gate.

5. (a) TTL output stage showing enabling/disabling input; **(b)** typical symbols.

TTL three-state output stage

The TTL output stage shown in *figure 5a* is from a standard TTL gate, but has been modified by the addition of an enabling/disabling input. When the applied signal to the enabling/disabling input is logic 1, the diode D_1 is reverse biased and therefore does not conduct. Similarly, the base-emitter junction (labelled B-E_2) does not conduct. The output stage of the gate

6



7

Control signals				Output	Comment
D	C	B	A		
0	0	0	0	0	No inputs to bus
0	0	0	1	Q_A	Gates B, C and D disconnected
0	0	1	0	Q_B	Gates A, C and D disconnected
0	1	0	0	Q_C	Gates A, B and D disconnected
1	0	0	0	Q_D	Gates A, B and C disconnected

therefore operates in the normal way. However, when the enabling/disabling input signal is logic 0, both transistors T_3 and T_4 are held open circuit, so the output terminal is effectively disconnected. Typical symbols are shown in *figure 5b*.

Examples of simple three-state bus systems

Figure 6 shows a simple bus system. Four 1-bit input devices are joined to the 1-bit bus by four three-state buffer gates (A, B, C and D) from the TTL 74126 IC. Each gate is enabled by a logic 1 control signal. As we know, only one gate may be enabled at any one time and the others must then be disabled. The possible enable/disable input signals along with the corresponding output and comments are shown in the truth table of *figure 7*. All other enable/disable input signal combinations are considered **illegal**, because they allow information from more than one input device to appear on the bus at the same time.

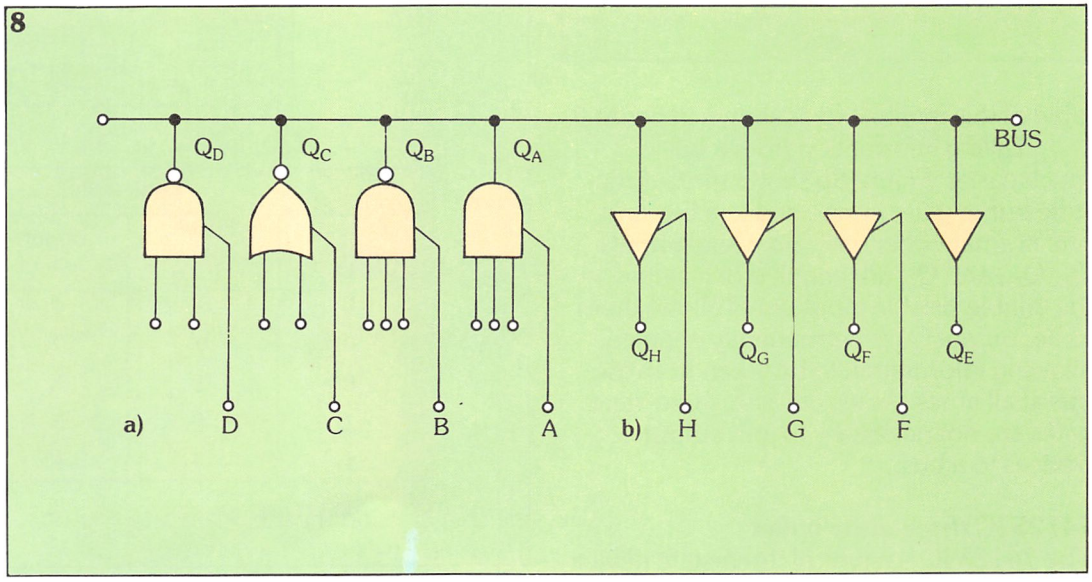
When the outputs of several three-state gates are joined on a bus, the effect is to produce a circuit which may select data. The enable/disable inputs perform the function of input selection as in a classic multiplexer. For example, four three-state gates may be used to make a 4-to-1 line data multiplexer, as in *figure 8a*. As shown, the three-state gates making up the multiplexer can be different varieties and have different numbers of data inputs. The truth

6. Simple bus system where four 1-bit input devices are joined to the 1-bit bus by four three-state buffer gates.

7. Truth table showing possible enable/disable inputs for the bus system shown in figure 6.

8. (a) Four, three-state gates being used to form a 4-to-1 line data multiplexer; **(b)** three, three-state buffer gates connected to a bus and one normal buffer gate.

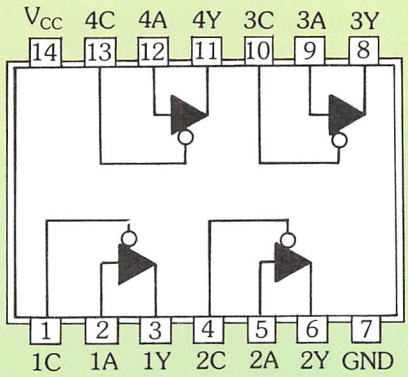
8



9

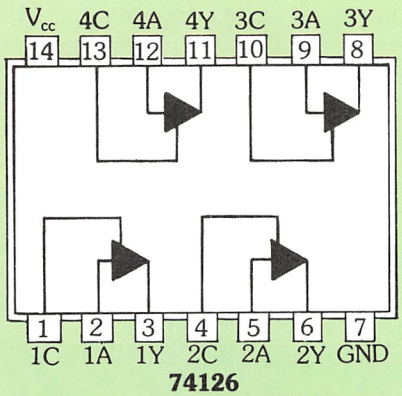
D	C	B	A	Output
0	0	0	1	Q_A
0	0	1	0	Q_B
0	1	0	0	Q_C
1	0	0	0	Q_D

10



74125

12

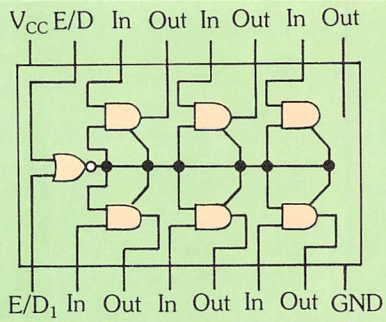


74126

11

First gate:	Input 1A,	output 1Y,	enable/disable	input 1C
Second gate:	Input 2A,	output 2Y,	enable/disable	input 2C
Third gate:	Input 3A,	output 3Y,	enable/disable	input 3C
Fourth gate:	Input 4A,	output 4Y,	enable/disable	input 4C

13



Truth table

Enable/ disable E/D_1	Enable/ disable E/D_2	Input data	Gate output
0	0	0	0
0	0	1	1
0	1	X	High impedance
1	0	X	High impedance
1	1	X	High impedance

(X = 'don't care')

table of this multiplexer is shown in *figure 9*.

Taking information *from* a bus is much easier. *Figure 8b* shows three three-state buffer gates connected to a bus and one normal buffer gate. So while outputs Q_F, Q_G, and Q_H are enabled only when the enable/disable input signal allows them to be, output Q_E is permanently enabled, allowing information to be taken from the bus at all times. As we can see, three-state gates are not necessary to join output devices to a bus.

74125 IC three-state buffer

The 74125 IC is a typical three-state device

Left: fault finding on components with a scanning electron microscope. (Photo: S.E.M.).

9. Truth table for the data multiplexer shown in figure 8a.

10. Pin configuration for the 74125 IC three-state buffer.

11. Four independent buffers of the 74125 IC shown in figure 10.

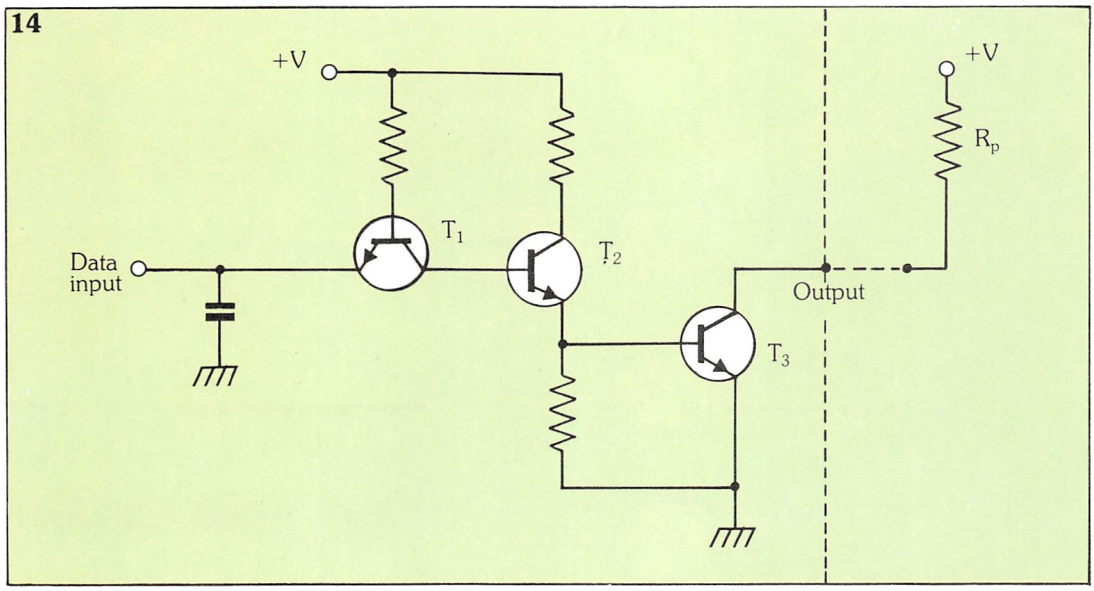
12. Pin configuration for the 74126 IC three-state buffer.

13. Pin configuration and truth table for the 8095 IC three-state buffer.

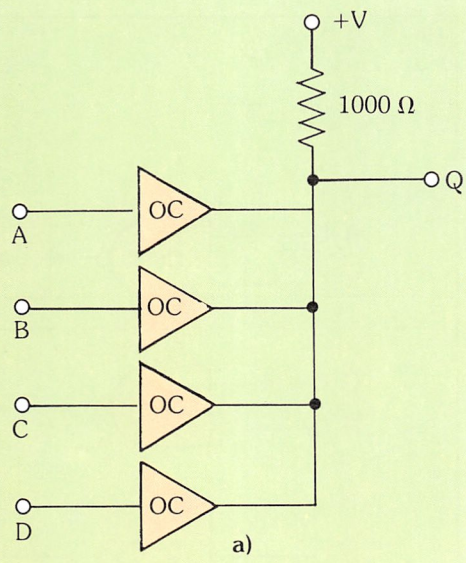
14. A buffer gate using open collector outputs.

15. (a) Circuit using four open collector buffers; (b) truth table for this circuit.

14



15



Truth table

D	C	B	A	Q
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

b)

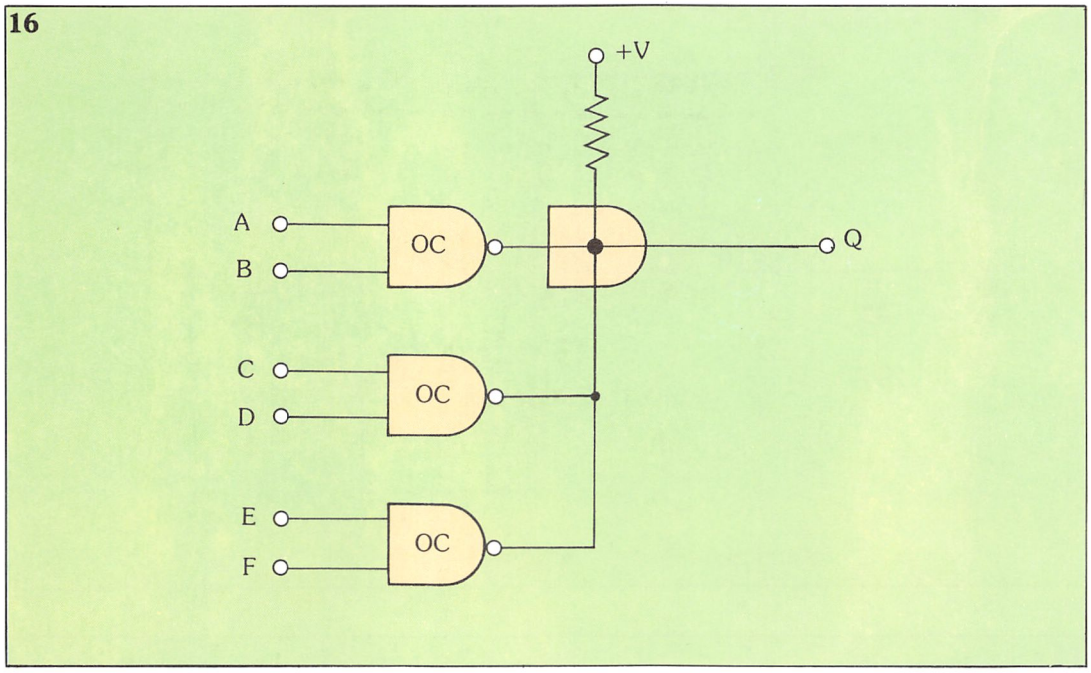
containing four separate three-state buffer gates, each with enable/disable inputs. Its pin configuration is shown in *figure 10* and the four independent buffers can be identified as shown in *figure 11*. The inverting circle, shown at the enable/disable inputs of the buffers, shows that each buffer is enabled by a logic 0 signal, and disabled by a logic 1.

74126 IC three-state buffer

Figure 12 shows the pin configuration for the 74126 IC and the four separate buffers can also be identified as shown in *figure 11*. Each buffer is enabled by a logic 1 signal at its enable/disable input and disabled by a logic 0.

8095 IC three-state buffer

The pin configuration, and a truth table relating enable/disable inputs and input data to output of the 8095 IC is shown in *figure 13*. The six, three-state AND gate buffers within the IC are enabled or disabled simultaneously by the output of a single two-input NOR gate. Thus, the inputs of the NOR gate form the controlling enable/disable inputs. The 8095 IC is



16. Using open collector output stages forms wired-OR logic.

popularly used to join input devices to busses of many microprocessor based computers.

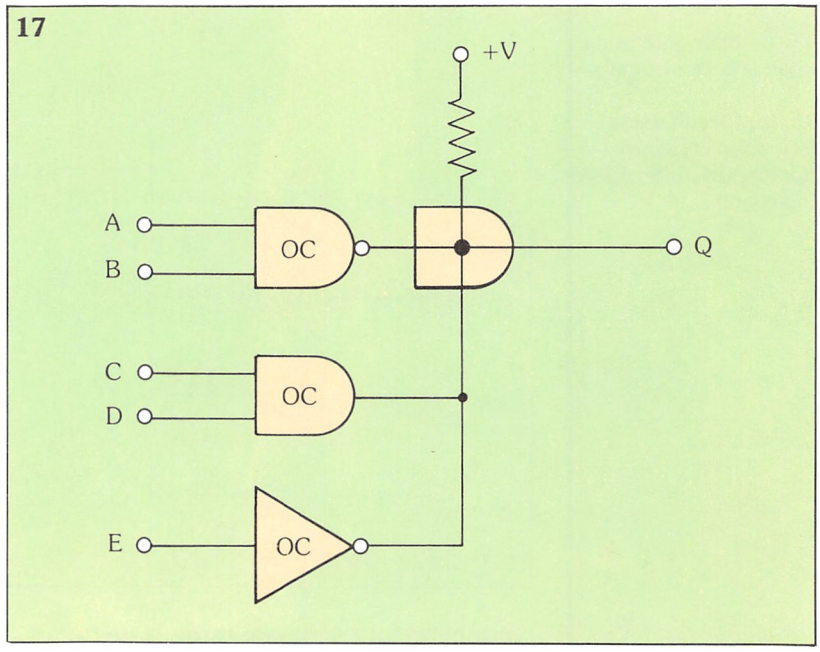
Open collector outputs stages

Another approach to the problem of joining several input devices onto a single bus is to use buffer gates with open collector outputs. In such gates (an example is shown in *figure 14*) the collector of the output transistor is not connected internally, i.e. it is left **open**. In use, the transistor collector is connected to +V (we say it is **tied** to +V) by a pull-up resistor, R_p in *figure 14*. Thus, when transistor T_3 is off, the collector voltage – the gate output – is +V, or logic 1.

Since the pull-up resistor is missing for all the open collector gates, they can all be connected together and tied to +V with a single pull-up resistor. For example, four open collector buffers are used in the circuit shown in *figure 15a*, where the single 1000 Ω resistor ties all the outputs to +5 V.

Circuit output Q, is logic 0 only if all the inputs are logic 0. If any one or more inputs is logic 1, the output Q also becomes logic 1 as shown in the truth table of *figure 15b*. Clearly, the circuit behaves like a four-input OR gate.

This **wired-OR logic**, which the use of open collector output stages forms, can



17. Three different gates with open collector output stages in a wired-OR circuit.

be shown diagrammatically as in *figure 16*. Here the outputs of three, two-input NAND gates are shown joined in a wired-OR circuit, where output:

$$Q = \overline{AB}.\overline{CD}.\overline{EF}$$
$$= \overline{AB + CD + EF}$$

Similarly *figure 17* shows three different gates with open collector output stages in a wired-OR circuit, where output

$$Q = \overline{AB}\,.\,CD\,.\,\overline{E}.$$

(continued in part 18)